Pós-Graduação em Ciência da Computação

“Um Catálogo de regras para transformação automática de esquemas EER em código SQL-Relacional: uma visão MDD com foco em restrições estruturais não triviais”

Por

Edson Alves Da Silva

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2015

EDSON ALVES DA SILVA

“Um Catálogo de regras para transformação automática de esquemas EER em código SQL-Relacional: uma visão MDD com foco em restrições estruturais não triviais "

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de mestre em Ciência da Computação.

ORIENTADOR: Prof. Dr. Robson do Nascimento Fidalgo

RECIFE
2015

Catalogação na fonte
Bibliotecário Jefferson Luiz Alves Nazareno CRB4-1758

S586c Silva, Edson Alves da.
Um catálogo de regras para transformação automática de esquemas EER em código SQL-relacional: uma visão MDD com foco em restrições estruturais não triviais. / Edson Alves da Silva. – Recife: O Autor, 2015.
146 f.: fig., tab.

Orientador: Robson do Nascimento Fidalgo.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIN, Ciência da Computação, 2015.
Inclui referências e apêndices.

1. Banco de dados. 2.Engenharia de software. 3. Algoritmos computacionais. 4. SQL (Linguagem de programação de computador) Fidalgo, Robson do Nascimento (Orientadora). II. Título.

005.74 CDD (22. ed.) UFPE-MEI 2015-120

Dissertação de Mestrado apresentada por **Edson Alves da Silva** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Um Catálogo de Regras para Transformação Automática de Esquemas EER em Código SQL-Relacional: uma Visão MDD com Foco em Restrições Estruturais não Triviais",** orientada pelo **Prof. Robson do Nascimento Fidalgo** e aprovada pela Banca Examinadora formada pelos professores:

_____

Prof. Jaelson Freire Brelaz de Castro
Centro de Informática / UFPE

_____

Prof. Maria Lencastre Pinheiro de Menezes Cruz
Escola Politécnica de Pernambco / UPE

_____

Prof. Robson do Nascimento Fidalgo
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 2 de março de 2015

_____

Profa. Edna Natividade da Silva Barros
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Agradecimentos

Primeiramente, agradeço a Deus pela realização desse mestrado e por tudo em minha vida. Agradeço à minha mãe Maria das Graças pela inspiração para superar todos os obstáculos. Aos meus irmãos: Eduardo, Ednaldo e Edjane que são indispensáveis em todos os momentos da minha vida. Agradeço também à minha noiva Caline Teixeira pela compreensão nos momentos de ausência.

Agradeço ao professor Robson Fidalgo pela orientação, pela confiança, pelo apoio, por sempre estar disponível para todas as dúvidas e pela motivação contínua nas discussões. Aos meus amigos, agradeço pelo apoio constante e amizade verdadeira. Em especial, agradeço a Natália, Antônio, Jackson, Jones e Tarcísio, que foram importantes para conclusão deste trabalho. Aos demais professores e funcionários do CIn/UFPE, agradeço por desempenharem tão bem as suas funções.

Por fim, agradeço ao CNPQ – Conselho Nacional de Desenvolvimento Cientifico e Tecnológico – por financiar este trabalho.

Resumo

Model Driven Development (MDD) é um paradigma para geração automática de código executável que utiliza modelos como o seu artefato primário. No escopo de Banco de Dados, apesar das regras para transformação de esquemas *Enhanced Entity Relationship* (EER) em código da *Structured Query Language* (SQL)-Relacional já terem sido amplamente exploradas na literatura, não se encontrou um trabalho que ao mesmo tempo especifique tradutores MDD capazes de transformar, automaticamente, esquemas EER em códigos SQL-Relacional e aborde restrições como: Participação em Relacionamento, Disjunção e Completude em Herança ou Categoria são transformadas em estruturas SQL-Relacional. Neste contexto, visando dar uma contribuição às limitações mencionadas, esta dissertação apresenta duas macros contribuições: 1) um Catálogo de regras para transformar um esquema EER em um esquema Relacional e este em código SQL; e 2) um algoritmo que especifica uma ordem correta para a execução automática destas regras. De modo a mostrar a viabilidade e aplicação prática deste trabalho, o Catálogo de regras de transformação e o algoritmo para automatização do Catálogo são codificados na linguagem *Query/View/Transformation-Relations* (QVT-R) e implementados na ferramenta EERCASE. A avaliação do trabalho foi feita a partir da transformação de esquemas EER (não triviais) em códigos SQL-Relacional, os quais são conferidos por especialistas de Banco de Dados. Por fim, comparando o trabalho proposto com os trabalhos relacionados investigados, constatou-se que o trabalho desta dissertação avança o estado da arte, pois é o único que é baseado em MDD e garante que as restrições de Participação em Relacionamento, Disjunção e Completude em Herança ou Categoria sejam automaticamente geradas para serem garantidas diretamente pelo Sistema de Gerenciamento de Banco de Dados.

Palavras Chave: Desenvolvimento Dirigido por Modelo. Projeto de Banco de Dados. Transformação de Modelo.

Abstract

Model Driven Development (MDD) is a paradigm for automatic generation of executable code that uses models as its primary artifact. In the database scope, despite the rules for transformation of Enhanced Entity Relationship (EER) schemas in code of Structured Query Language (SQL)-Relational have already been widely explored in the literature, we did not find a work that, at the same time, specifies MDD translators capable of transforming, automatically, EER schemas in SQL-Relational codes and addresses restrictions such as: Participation in Relationship, Disjunction and Completeness in Inheritance or Category are transformed into SQL-relational structures. In this context, in order to contribute for the mentioned limitations, this dissertation presents two macro contributions: 1) a rule Catalog to transform an EER schema into a Relational schema and this SQL code; and 2) an algorithm that specifies a correct order for the automatic enforcement of these rules. In order to show the feasibility and practical application of this work, the Catalog of transformation rules and the algorithm for Catalog automation are encoded in Query/View/Transformation-Relations (QVT-R) language and implemented in EERCASE tool. The evaluation of the work was made from the processing of EER schemas (non-trivial) in SQL-Relational codes, which are conferred by database experts. Finally, comparing the proposed work with the related work investigated, it was found that the proposed work advances the state of the art, as it is the only one that is based on MDD and ensures that the restrictions on Participation in Relationship, Disjunction in Inheritance and Completeness in Inheritance or Category are guaranteed by the Database Management System.

Keyword: Model Driven Development. Database Design. Model Transformation.

Lista de Figuras

Lista de Quadros

Lista de Algoritmos

Principais Acrônimos

BD	Banco de Dados
CASE	Computer-Aided Software Engineering
DSL	Domain-Specific Language
EER	Enhanced Entity Relationship
EERCASE	Enhanced Entity Relationship CASE
EERMM	Enhanced Entity Relationship MetaModel
EGL	Epsilon Generation Language
EMP	Eclipse Modeling Project
EMF	Eclipse Modeling Framework
IMM	Information Management Metamodel
M2M	Model to Model
M2T	Model to Text
MDD	Model Driven Development
MOF	Meta-Object Facility
OMG	Object Management Group
QVT	Query View Transformation
SGBD	Sistema Gerenciador de Banco de Dados
UML	Unified Modeling Language
SQL	Structured Query Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

Sumário

1 Introdução

Este capítulo contextualiza e apresenta o problema de pesquisa que motivou o desenvolvimento deste trabalho, os objetivos a serem alcançados, o método para atingir estes objetivos e a estrutura dos demais capítulos desta dissertação.

1.1 Contextualização

No paradigma de desenvolvimento dirigido por modelo (do inglês *Model-Driven Development* - MDD), modelos são mais que meros artefatos de documentação, pois estes são considerados como objetos executáveis. Em outras palavras, uma vez que um modelo é criado segundo as regras de boa formação da sua linguagem de modelagem, este pode ser transformado automaticamente em outro modelo (i.e., transformação modelo-modelo ou M2M) ou em código executável/interpretável (i.e., transformação modelo-texto ou M2T) (KELLY; TOLVANEN, 2007). Ou seja, dado que o modelo deve estar em conformidade com a sintaxe e a semântica de sua linguagem de modelagem, este não é um simples desenho (BRAMBILLA; CABOT; WIMMER, 2012). Posto isto, no contexto desta dissertação, o paradigma MDD foi escolhido pelo fato de permitir que as transformações do tipo M2M e M2T sejam feitas de forma automática.

O projeto de Banco de Dados (BD) (ELMASRI; NAVATHE, 2010)(SILBERSCHATZ; KORTH; SUDARSHAN, 2010)(DATE, 2003), tradicionalmente, é dividido em três fases: projeto conceitual, projeto lógico e projeto físico. Cada fase com o seu modelo de dados específico. A fase de projeto conceitual é o nível mais abstrato, uma vez que é independente de tecnologia e, consequentemente, é próximo ao entendimento do usuário, o que facilita a coleta da semântica do domínio a ser modelado. No projeto lógico, o modelo de dados do projeto conceitual é refinado/transformado para um modelo mais computável. Por fim, no projeto físico, o modelo de dados do projeto lógico é enriquecido com detalhes específicos da tecnologia em que o

BD é implementado. Dado que o projeto conceitual é a primeira das três fases do projeto de BD, é importante que os conceitos desta fase sejam corretamente utilizados, pois erros nesta fase são propagados para as demais, o que torna as correções mais complexas e custosas (PARENT; SPACCAPIETRA; ZIMÁNYI, 1999).

Existem várias notações que podem ser usadas para modelar a fase conceitual do BD (SONG; EVANS; PARK, 1995). Algumas destas permitem somente Relacionamento binário e por esse motivo são conhecidas como notações binárias, enquanto outras permitem Relacionamentos n-ários e por isto são chamadas notações n-árias. No geral, a maioria das ferramentas para diagramar esquemas conceituais de BD usa notações binárias. No entanto, estas notações 1) não permitem Atributos em Relacionamentos – o que não parece natural, pois Relacionamentos podem ter propriedades descritivas; e 2) não permitem o Relacionamento entre três ou mais Entidades – o que é um inconveniente, pois existem Relacionamentos que são intrinsecamente n-ários e transformar um Relacionamento n-ário em *n* binários gera uma sobrecarga visual, uma vez que se utiliza mais elementos gráficos. Independente da notação usada, existem duas convenções diferentes para se referir ao lugar onde Papéis, Cardinalidades[1] e Participações[2] são especificadas em um Relacionamento, a saber, "*look-across*" (i.e., lado oposto da Entidade) e "*look-here*" (i.e., mesmo lado da Entidade) (HARTMANN, 2003) (SONG; EVANS; PARK, 1995).

Dentre as notações para o projeto conceitual de BD, o Modelo Entidade Relacionamento Estendido (do inglês *Enhanced Entity-Relationship* - EER) (CHEN, 1976) (ELMASRI; NAVATHE, 2010) e o Diagrama de Classes (DC) da Linguagem Unificada de Modelagem (do inglês *Unified Modeling Language*-UML) (RUMBAUGH; JACOBSON; BOOCH, 2004) são as mais usadas. Neste contexto, três estudos (DE LUCIA et al., 2009) (DE LUCIA et al., 2008) (BAVOTA et al., 2011) fazem comparações entre o Modelo EER e o DC, reportando resultados interessantes. (DE LUCIA et al., 2009) e (DE LUCIA et

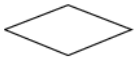
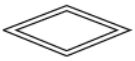
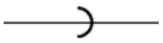
[1] Número máximo de ocorrências que uma instância de uma Entidade pode ter no Relacionamento.
[2] Define se todas as instâncias de uma Entidade estão relacionadas com pelo menos uma instância da mesma ou de outra Entidade.

al., 2008) sugerem que a notação do DC é geralmente mais compreensiva do que a notação do EER. No entanto, (BAVOTA et al., 2011) conclui que o entendimento do EER é significativamente maior que o DC, se “Atributos compostos”, “Atributos multivalorados” e “Entidades Fracas” são requeridos em uma dada modelagem. Ressalta-se que os autores ainda não consideraram construções como “Entidades Associativas”, “Atributos em Relacionamento” e a “imprecisão de notações *look-across* para relacionamento n-ários”. Por esse motivo, alguns construtores importantes do Modelo EER ainda precisam ser avaliados. Posto isto, pode-se concluir que qualquer comparação entre o Modelo EER e o DC pode fornecer resultados para debate e discussão, sendo a escolha de uma notação em vez de outra uma questão de preferência ou de contingência do projeto.

Dada a discussão acima, este trabalho adotará o Modelo EER para a fase conceitual, pois este é bem aceito pela comunidade de BD, haja vista que: 1) muitos pesquisadores usam o Modelo EER (DAVIES et al., 2006) (BOLLATI et al., 2012) (AMALFI et al., 2011) (MOTTA; PIGNATELLI, 2011) (BADIA; LEMIRE, 2011); 2) existem ferramentas CASE (do inglês Computer Aided Software Engineering) que auxiliam a modelagem EER (e.g., EER/Studio XE2, PowerDesigner, ERWin, SmartDraw e EERCASE); 3) muitos livros dedicam um ou mais capítulos sobre o assunto (ELMASRI; NAVATHE, 2010), (GARCIA-MOLINA; ULLMAN; WIDOM, 2008), (KROENKE; AUER, 2013), (TEOREY et al., 2011), (SILBERSCHATZ; KORTH; SUDARSHAN, 2010), (RAMAKRISHNAN; GEHRKE, 2002), (DATE, 2003) e (HEUSER, 1998); 4) o ensino do EER é recomendado pelos Currículos de Computação da ACM/IEEE/AIS (ACM/IEEE, 2008) (ACM/AIS, 2010); 5) é uma notação n-ária; e 6) em se tratando de projeto conceitual, de acordo com Bovota et. al. (BAVOTA et al., 2011), EER é mais expressivo do que a alternativa de usar o DC. Neste contexto, dado que, de acordo com (SONG; EVANS; PARK, 1995), existem dez notações para o modelo EER (e.g., 1) Chen, 2) Teorey, 3) Elmasri & Navathe, 4) Korth & Silberschatz, 5) McFadden & Hoffer, 6) Batine, Ceri, e Navathe, 7) Oracle's CASE*METHOD, 8) Information Engineering, 9) IDEF1X Information Model Notation e 10) Bachman), este trabalho baseado-se na notação de Elmasri e Navathe (ELMASRI; NAVATHE, 2010), pois esta é

próxima à notação clássica de Chen e é bem aceita pela comunidade de BD (SONG; CHEN, 2009). Sobre a notação de Elmasri e Navathe destaca-se que a leitura da Participação é look-here e da Cardinalidade é look-across. Na Figura 1, apresenta-se um fragmento da notação do Modelo EER proposta por Elmasri e Navathe (mais detalhes na Seção 2.1.3).

Figura 1 - Fragmento da notação do Modelo EER por Elmasri e Navathe.

Construtores EER	Descrição
	Entidade Regular
	Entidade Fraca
	Atributo Comum
	Atributo Identificador
	Relacionamento
	Relacionamento Identificador
	Participação (Parcial e Total)
1 N	Cardinalidade (1 ou N)
	Herança Direta

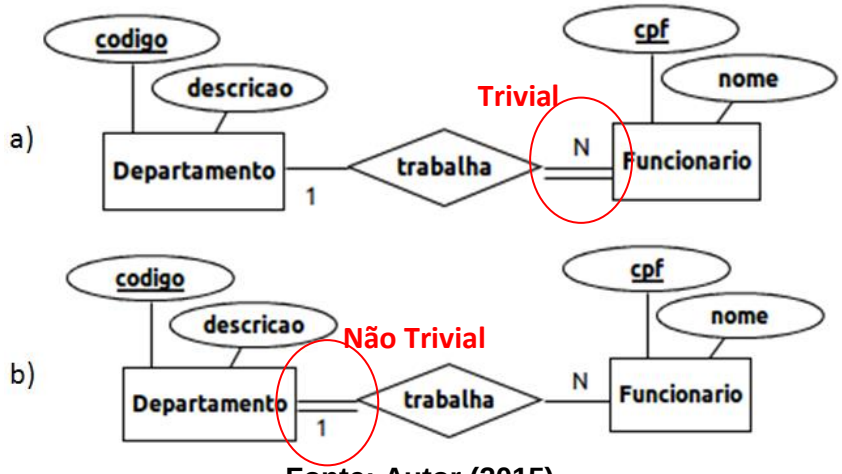
Fonte: Adaptado de Elmasri e Navathe (2010).

1.2 Problema e Motivação

De modo simplificado, a transformação de um esquema EER em um esquema Relacional (CODD, 1970) dá-se a partir do mapeamento de Entidades, Atributos e Relacionamentos em Tabelas, Colunas, Chaves Primárias e Chaves Estrangeiras. Contudo, existem restrições do Modelo EER (e.g., Participação Total-Total, Herança Disjunta e Categoria Total) que exigem o uso de recursos relacionais não triviais (i.e., Gatilhos, Asserções e/ou Cheques em BD). Assim, neste trabalho, considera-se como “não trivial” toda restrição EER que, para ser transformada em uma construção relacional, exige recursos como Gatilhos, Asserções e/ou Cheques. Na Figura 2, apresenta-se um exemplo que faz uso de restrições EER trivial (Figura 2-a) e não trivial (Figura 2-b). Destaca-se que as regras não triviais podem ser especificadas

como regras de negocio de um sistema, sendo estas implementadas diretamente no código na aplicação.

Figura 2 - Esquemas EER com restrição estrutural de Participação trivial e não trivial.



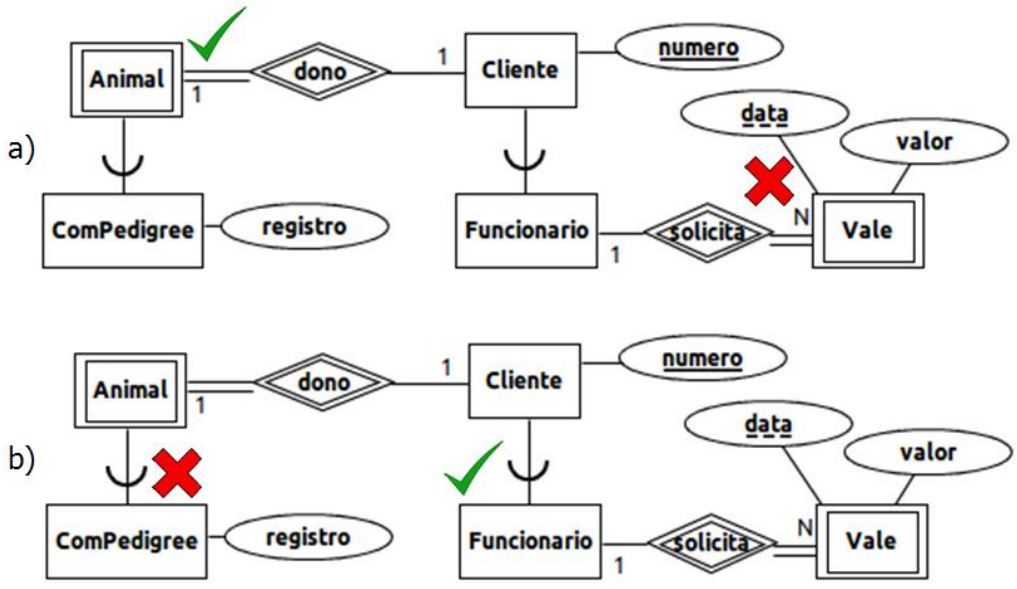
Fonte: Autor (2015).

Na Figura 2, note que apesar de a cardinalidade dos esquemas “a” e “b” serem iguais (i.e., um funcionário trabalha em um único departamento, no qual podem trabalhar vários funcionários), a Participação dos dois esquemas é diferente. No esquema “a”, a Participação é definida como Total apenas para Funcionário, enquanto no esquema “b” é especificada como Total apenas para Departamento. No esquema “a”, todo registro de Funcionário deve estar relacionado com um registro de Departamento. Neste caso, para assegurar esta regra de negócio, basta definir a Chave Estrangeira de Departamento em Funcionário como nulo que o SGBD garante, ao inserir, excluir ou atualizar um registro de Funcionário, que este registro sempre estará vinculado com um registro de Departamento. Por sua vez, no esquema “b”, tem-se que todo registro de Departamento deve estar vinculado com ao menos um registro de Funcionário. Diferente do caso anterior, para garantir que todo registro de Departamento, ao ser inserido, excluído ou alterado deve estar relacionado com ao menos um registro de Funcionário, é necessário a definição de dois Gatilhos. O primeiro para garantir que ao inserir um registro em Departamento deve existir um registro em Funcionário com um valor de Chave Estrangeira apontando para o Departamento que está sendo inserido. Por sua vez, o segundo gatilho para garantir que ao atualizar um valor da Chave Estrangeira

de Departamento em Funcionário ou excluir um registro de Funcionário, todos os registros de Departamento continuarão associados com ao menos um Funcionário.

Outro problema que motiva o desenvolvimento deste trabalho refere-se à dependência estrutural de identificador que existe entre construtores EER, pois este tipo de dependência influencia na ordem de transformação das estruturas EER em estruturas Relacionais. Por exemplo, o identificador de uma Entidade Fraca é formado a partir do identificador da sua entidade Forte e o identificador de uma Sub-entidade deve ser o mesmo da sua Super-entidade. Assim, uma Entidade Fraca só pode ser transformada depois da sua Entidade Forte e uma Sub-entidade depois da sua Super-entidade. Considerando este contexto e o exemplo apresentado na Figura 3, pode-se constatar que existem situações de impasse que não podem ser transformadas a partir de uma sequencia de execuções em cascata (e.g., transformar todas as Entidades Fracas e só depois as Sub-entidades ou vice-versa).

Figura 3 - Dependências estruturais entre construtores do Modelo EER.



Fonte: Autor (2015).

Na Figura 3-a, considera-se que a transformação das Entidades ocorre na seguinte ordem/cascata: primeiro Entidade Regular, depois Entidade Fraca e, por fim, Sub-entidade. Dada esta sequencia, no exemplo a Entidade Vale é Entidade Fraca de *Funcionário*, e tem dependência do seu identificador. Assim,

Vale não pode ser transformada (✖) primeiro que *Funcionário*, pois Sub-entidades só podem ser transformadas depois que todas as Entidades Fracas tiverem sido transformadas. Mantendo este raciocínio, entretanto, mudando a sequencia das transformações (i.e., primeiro Entidade Regular, depois Sub-entidade e, por fim, Entidade Fraca), na Figura 3-b, tem-se o mesmo impasse com a transformação da Entidade *ComPedigree*. Isto é, a Sub-entidade *ComPedigree* não pode ser transformada, pois esta depende que a Entidade Fraca *Animal* já tenha sido transformada, o que viola a sequencia/cascata especificada.

Dados os problemas apresentados (i.e., restrições estruturais não triviais e dependências estruturais de identificadores) e que o uso de MDD/Transformação de Modelos é um tema de pesquisa ainda pouco explorado no contexto de projeto de Banco de Dados, a motivação para o desenvolvimento deste trabalho é: se desconhece a existência de outro trabalho que, à luz de MDD e considerando as restrições não triviais e as dependências estruturais de identificador do Modelo EER, defina como transformar, automaticamente e sem perda de semântica, todos os construtores do Modelo EER (segundo a notação de (ELMASRI; NAVATHE, 2010)) em construtores do Modelo Relacional.

1.3 Objetivo

Considerando o paradigma de MDD e o contexto em questão (i.e., restrições não triviais e dependências de identificador do Modelo EER), o objetivo geral deste trabalho é garantir que o esquema fisíco implementado reflita a semântica especificada no esquema conceitual, evitando que erros sejam cometidos pelo projetista no projeto de BD. Este objetivo é alcançado com a automatização da transformação de esquemas EER em esquemas Relacionais. A partir do objetivo geral, podem-se descrever os seguintes objetivos específicos:

1. Identificar quais são as construções EER que geram restrições não triviais e quais têm dependências estruturais de identificadores;

2. Especificar, em linguagem algorítmica, um Catálogo de Regras para transformar construções EER em construções Relacionais;

3. Definir um algoritmo que, dadas as restrições não triviais e as dependências estruturais de identificador do Modelo EER, seja capaz de automatizar a execução do catálogo proposto;

4. Implementar e avaliar as propostas apresentadas neste trabalho a partir de uma ferramenta CASE.

1.4 Método

Os métodos de pesquisa na Engenharia de Software podem ser classificados em quatro tipos, a saber: científico, da engenharia, empírico e analítico (TRAVASSOS; GUROV; AMARAL, 2002). O método de engenharia caracteriza-se por ser uma abordagem orientada à melhoria evolutiva que assume a existência de algum modelo do processo ou produto de software e modifica este modelo com propósito de melhorar os objetos do estudo. Este trabalho se baseia no método de engenharia, pois observa as soluções existentes, sugere uma solução mais adequada, que é desenvolvida e analisada.

De acordo com o método, a pesquisa foi realizada em três fases:

1. **Revisão de literatura**: realizado um estudo detalhado sobre os conceitos, características, trabalhos realizados, problemas e possíveis soluções em transformação de esquemas EER em esquemas Relacionais e este em Código SQL-Relacional;

2. **Processo de design**: planejamento e desenvolvimento do Catálogo de regras para transformação de esquemas EER em Relacionais, com base no processo de transformação MDD;

3. **Avaliação**: realização de entrevistas semi-estruturas com especialistas de Banco de Dados, a fim de apresentarção de uma visão crítica sobre os dados coletados.

1.5 Estrutura da Dissertação

Visando atingir os objetivos desta dissertação, bem como apresentar conceitos e tecnologias relacionadas a este trabalho, os demais capítulos estão organizados conforme descrito a seguir. O Capítulo 2, apresenta os conceitos básicos e as tecnologias relacionadas a este trabalho. O Capítulo 3, avalia os principais trabalhos relacionados, destacando suas vantagens e limitações. O Capítulo 4, apresenta como os três primeiros objetivos deste trabalho são alcançados; Isto é, a definição e classificação das principais dependências estruturais do Modelo EER, a proposta do Catálogo de regras de transformação e a especificação do algoritmo para automatizar as regras do Catálogo em questão. O Capítulo 5, mostra-se como o último objetivo deste trabalho é realizado; Ou seja, como o Catálogo é implementado em uma ferramenta CASE e como o trabalho realizado foi avaliado. O Capítulo 6 apresenta as conclusões desta dissertação, suas principais contribuições, bem como as indicações de trabalhos futuros. Os Apêndice A e B apresentam os códigos necessário para implementação das transformações. Por fim, o Apêndice C apresenta os códigos gerados a partir das transformações.

2 Fundamentação Teórica

Neste Capítulo são apresentados os principais conceitos necessários para o entendimento dos próximos capítulos desta dissertação, a saber: Desenvolvimento Dirigido por Modelo, Transformação de Modelo, Modelo e Metamodelo EER e Modelo e Metamodelo Relacional.

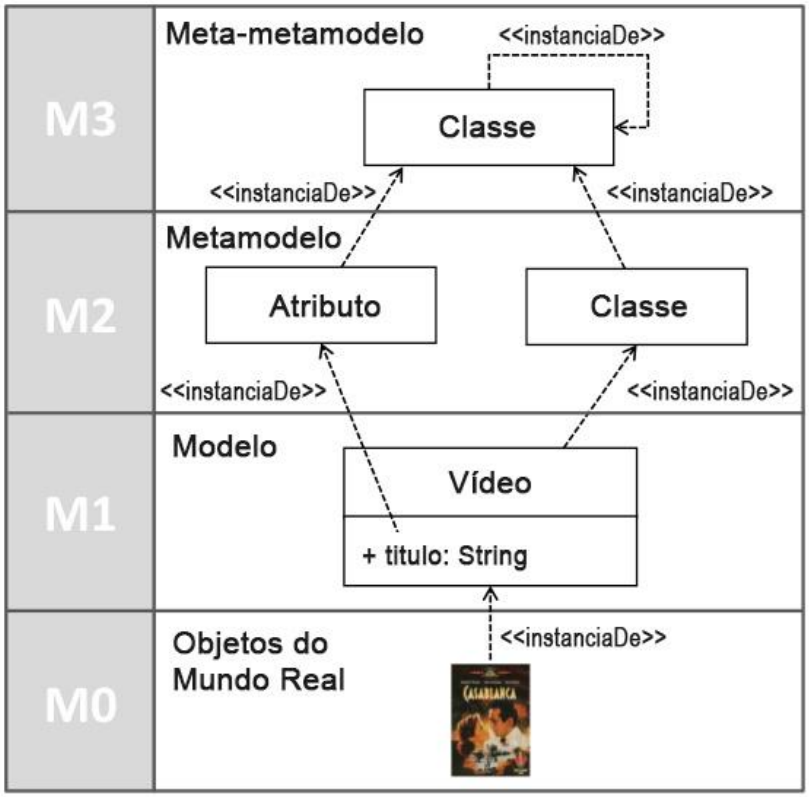
2.1 Desenvolvimento Dirigido por Modelo

Desenvolvimento Dirigido por Modelo é definido como um paradigma de desenvolvimento de *software* que tem modelos como artefatos principais em todas as fases do desenvolvimento (BRAMBILLA; CABOT; WIMMER, 2012). Modelos podem capturar informações de diferentes fases do processo de desenvolvimento, como requisitos, projeto, implementação, testes, analise de qualidade, simulação e verificação (CHO; GRAY; SYRIANI, 2012). Em MDD, modelos são mais do que documentação, eles são objetos executáveis. Em outras palavras, uma vez que um modelo é criado, código correspondente pode ser gerado e então compilado ou interpretado para execução. Deste modo, MDD esconde detalhes de implementação, o que aumenta a velocidade do processo de desenvolvimento, pois reduz a complexidade (KELLY; TOLVANEN, 2007).

No contexto de MDD, modelo é uma abstração dos fenômenos e objetos do mundo real. Por sua vez, o metamodelo é a descrição dos conceitos do modelo e como estes conceitos podem ser relacionados. Desta forma, podem-se definir modelos do mundo real e, em seguida, os modelos que descrevem estes modelos – denominados metamodelos – e, recursivamente, modelos que descrevem os metamodelos – denominados meta-metamodelos. Embora, na teoria, podem-se definir níveis infinitos de metamodelos, na prática, meta-metamodelos podem ser definidos com base em si mesmos e, desta forma, não é necessário ir além deste nível de abstração. Em qualquer nível, dizemos que um modelo está em conformidade com o seu metamodelo, da mesma forma que um programa de computador está em conformidade com a

gramática da linguagem de programação em que está escrito (e.g., EBNF) (BRAMBILLA; CABOT; WIMMER, 2012).

Figura 4 - Modelos, metamodelos e meta-metamodelos.



Fonte: (BRAMBILLA; CABOT; WIMMER, 2012)**.**

Na Figura 4, apresenta-se um exemplo de instância, modelo, metamodelo e meta-metamodelo em quatro níveis: 1) No nível M0 estão os objetos do mundo real, neste caso, um filme; 2) No nível M1 está a representação modelada, onde é descrito o conceito de vídeo com seus atributos, neste caso, um título; 3) No nível M2 está o metamodelo, que descreve o modelo do nível M1, com os conceitos de Classe, Atributo e Instância; e 4) No nível M3 está representado o meta-metamodelo, que define os conceitos utilizados no nível M2.

2.1.1 Transformação de Modelo

Transformação de Modelo é um conceito central em MDD, uma vez que fornece mecanismos para automatizar a manipulação de modelos (e.g., a transformação de um diagrama de Classes da UML em um diagrama ER).

Assumindo que a Transformação de Modelo é corretamente definida, também pode ser usada para garantir a consistência entre modelos (HUBER, 2008).

No contexto de Transformação de Modelo, define-se que: 1) a descrição da transformação, expressa como um modelo origem, é transformado em um modelo destino; e 2) a regra de transformação descreve como um fragmento do modelo origem pode ser transformado em um fragmento do modelo destino. Uma regra de transformação contém um padrão de origem e um padrão de destino. Para cada ocorrência do padrão de origem no modelo de origem, um padrão de destino é criado no modelo de destino (BIEHL, 2010)(MENS; GORP, 2006). Neste contexto, Kleppe (KLEPPE; WARMER; BAST, 2003) define Transformação de Modelo como a geração automática de um modelo destino a partir de um modelo origem, de acordo com a descrição da transformação. Um modelo destino é a saída da transformação e deve estar em conformidade com o metamodelo destino.

Transformações de Modelo podem mudar o nível de abstração entre os modelos origem e destino da transformação, podendo esta transformação ser vertical ou horizontal. Na transformação vertical é modificado o nível de abstração do modelo e na transformação horizontal é alterada a representação do modelo. Além disso, a Transformação de Modelo pode ser endógena (i.e., o metamodelo origem e destino são os mesmos) ou exógena (i.e, a Transformação de Modelo mapeia conceitos entre diferentes metamodelos) (MENS; GORP, 2006) (CZARNECKI; HELSEN, 2003).

Uma Transformação de Modelo é distinguida quanto ao tipo de destino, podendo esse destino ser um modelo ou texto. Uma transformação modelo para modelo (M2M) cria elementos do modelo destino (i.e., elementos do modelo origem são mapeados para elementos do modelo destino) e uma transformação modelo para texto (M2T) cria texto/código (i.e., elementos do modelo origem são arbitrariamente fragmentados como texto). Uma vez que o texto carece de estruturas, transformações M2T são mais difíceis de analisar (BIEHL, 2010) (JILANI; USMAN; HALIM, 2010). A seguir, os principais Modelos para BD (i.e., Modelos EER e Relacional) são resumidamente apresentados.

2.1.2 Modelos e Metamodelos para Banco de Dados

No contexto de uma linguagem, um modelo define a sintaxe concreta (ou notação gráfica) e o metamodelo define a sintaxe abstrata (ou gramática) e a semântica (ou interpretação dos conceitos). Nas seções a seguir são apresentados os Modelos EER e Relacional, bem como seus respectivos metamodelos, o *Enhanced Entity Relationship MetaModel* (EERMM) e o IMM Relacional.

2.1.3 Modelo EER

Dentre os modelos de dados que podem ser usados para especificar esquemas de BD no nível conceitual, o modelo Entidade-Relacionamento Estendido (EER) (ELMASRI; NAVATHE, 2010) (SILBERSCHATZ; KORTH; SUDARSHAN, 2010) (RAMAKRISHNAN; GEHRKE, 2002) (TEOREY et al., 2011) é um dos mais expressivos e utilizados pela comunidade de BD (BAVOTA et al., 2011) (FIDALGO et al., 2012). O Modelo EER consiste basicamente de uma coleção de Entidades, Relacionamentos entre essas Entidades e Atributos. Estes conceitos básicos, bem como outros avançados, são resumidos a seguir. Uma explicação detalhada sobre os conceitos pode ser encontrada em (ELMASRI; NAVATHE, 2010) (SILBERSCHATZ; KORTH; SUDARSHAN, 2010) (RAMAKRISHNAN; GEHRKE, 2002) (TEOREY et al., 2011).

- Entidade Regular representa algo tangível (e.g., um carro ou um funcionário) ou intangível (e.g., uma conta bancária ou um curso de graduação) da realidade modelada;

- Relacionamento representa uma associação entre instâncias de Entidades. Todo Relacionamento tem Grau, Cardinalidade, Participação e Papel. Contudo, somente alguns têm Papais e são definidos como fracos (ou identificadores). Isto é:

    - Grau é a quantidade de Entidades envolvidas no Relacionamento. Por exemplo, unário (só uma Entidade – também conhecido como Auto Relacionamento), binário (duas Entidades) e N-ário (três ou mais Entidades);

- Cardinalidade especifica o número máximo de ocorrências que uma instância de uma Entidade pode ter no Relacionamento. A Cardinalidade pode ser representada por qualquer número inteiro e positivo. Contudo, convencionou-se usar os valores 1 e N para representar a Cardinalidade um ou vários, respectivamente;

- Participação define se todas as instâncias de uma Entidade estão relacionadas com pelo menos uma instância da mesma ou de outra Entidade (i.e. Participação Total) ou não (i.e., Participação Parcial). A Participação pode ser representada por qualquer número natural. Contudo, convencionou-se o valor 0 (Participação Parcial) e 1 (Participação Total). Ressalta-se que este conceito também é conhecido como Cardinalidade mínima;

- Papel representa a função que uma Entidade desempenha no Relacionamento (e.g., uma instância de um empregado pode ter o papel de chefe ou de subordinado em um Auto Relacionamento).

- Atributo representa uma propriedade de uma Entidade ou de um Relacionamento, podendo um Atributo ser uma composição dos seguintes tipos: simples ou composto, monovalorado ou multivalorado, simples, derivado, discriminador ou identificador. A seguir, esses tipos são resumidamente explicados e exemplificados. Destaca-se que um Atributo pode ser de mais de um tipo (e.g., simples e multivalorado) e que ele depende do contexto a ser modelado e não do nome do Atributo (e.g., endereço, dependendo do contexto, pode ser um Atributo simples ou composto).

  - Simples: define que o Atributo é indivisível (e.g., telefone e endereço);

  - Composto: define que o Atributo é divisível em subpartes (e.g., telefone dividido em prefixo e sufixo e endereço dividido em principal e complemento);
  - Monovalorado: define que o atributo pode ter apenas um valor por instância da Entidade ou Relacionamento (e.g., CPF e Identidade);
  - Multivalorado: define que o Atributo pode ter mais de um valor por instância da entidade (e.g., telefone e endereço);
  - Derivado: define que o Atributo não será armazenado e terá o seu valor definido a partir de cálculos sobre outros Atributos;
  - Identificador (ou chave): define que cada valor do Atribuo é único e usado como principal chave de acesso a uma instância da Entidade;
  - Discriminador (ou chave parcial): define que o Atributo funciona como um identificador parcial de cada instância da Entidade.

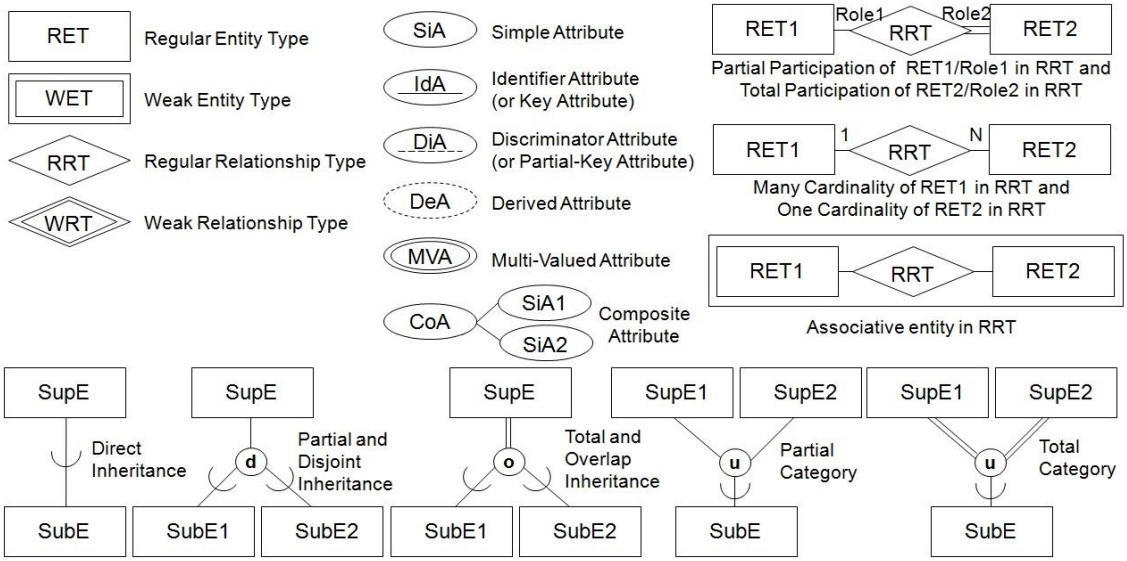
- Entidade Fraca expressa que uma Entidade necessita do identificador de outra Entidade para compor o seu identificador;
- Relacionamento Identificador denota um Relacionamento entre uma Entidade Forte e uma Entidade Fraca, no qual a Entidade Fraca necessita do identificado da Entidade Forte para compor seu identificador;
- Entidade Associativa (ou Agregação) corresponde a um Relacionamento que também pode ser entendido como uma Entidade;
- Herança é uma estrutura de modelagem que permite especializar ou generalizar uma Entidade. Isto é, definir uma Sub-Entidade (especialização) ou uma Super-Entidade (generalização). Uma Herança pode ser classificada como Direta quando existe somente

uma Super-Entidade e uma Sub-Entidade. Além disso, uma Herança pode ter dois tipos de restrição: de Disjunção (i.e., Disjunta ou Sobreposta) e de Completude/Participação (i.e., Parcial ou Total).

- Disjunção Disjunta quando as instâncias das Sub-Entidades devem ser mutuamente exclusivas;
- Disjunção Sobreposta especifica que as instâncias da Super-Entidade podem ser pertencer a mais de uma Sub-Entidade;
- Completude Total quando toda instância da Super-Entidade é especializada em, pelo menos, uma Sub-Entidade;
- Completude Parcial quando existem instâncias da Super-Entidade que não são nenhuma Sub-Entidade;

- Categoria representa um tipo de Herança seletiva, em que a Sub-Entidade chama-se Categoria e subconjunto da união das suas Super-Entidades. Uma Categoria sempre é Disjunta, mas, assim como Herança, uma Categoria pode ter Completude Total ou Parcial. Ressalta-se que uma Categoria com Completude Total é equivalente a uma Herança Disjunta Total (ELMASRI; NAVATHE, 2010).

Figura 5 - Construtores do Modelo EER.



Fonte: (ELMASRI; NAVATHE, 2010)**.**

Na Figura 5, mostram-se os construtores do Modelo EER segundo a notação de Elmasri e Navathe (ELMASRI; NAVATHE, 2010). Esta notação foi escolhida por ser n-ária, usar a abordagem *Look Here* para especificar Papel e Participação e *Look Across* para especificar Cardinalidade. Isto é, a restrição de Participação é definida perguntando-se para a Entidade de origem se todas as suas instâncias devem estar associadas com pelo menos uma ocorrência do Relacionamento. Por sua vez, a restrição de Cardinalidade é definida perguntando-se para a Entidade de destino qual é a quantidade máxima de ocorrências que ela pode ter no Relacionamento. Além disso, está notação é mais próxima da notação clássica do modelo ER de Chen, como também é uma das notações mais usadas pela comunidade de BD (DAVIES et al., 2006) (BOLLATI et al., 2012) (AMALFI et al., 2011) (MOTTA; PIGNATELLI, 2011) (BADIA; LEMIRE, 2011).

2.1.4 Metamodelo EERMM

O EERMM (FIDALGO et al., 2012) (FIDALGO et al., 2013) é um metamodelo que específica os construtores do Modelo EER e como estes podem ser relacionados. Vale ressaltar que desde a proposta do modelo ER (CHEN, 1976), o EERMM é a primeira solução que cobre todos os conceitos do modelo EER segundo a notação de Chen/Elmasri e Navathe.

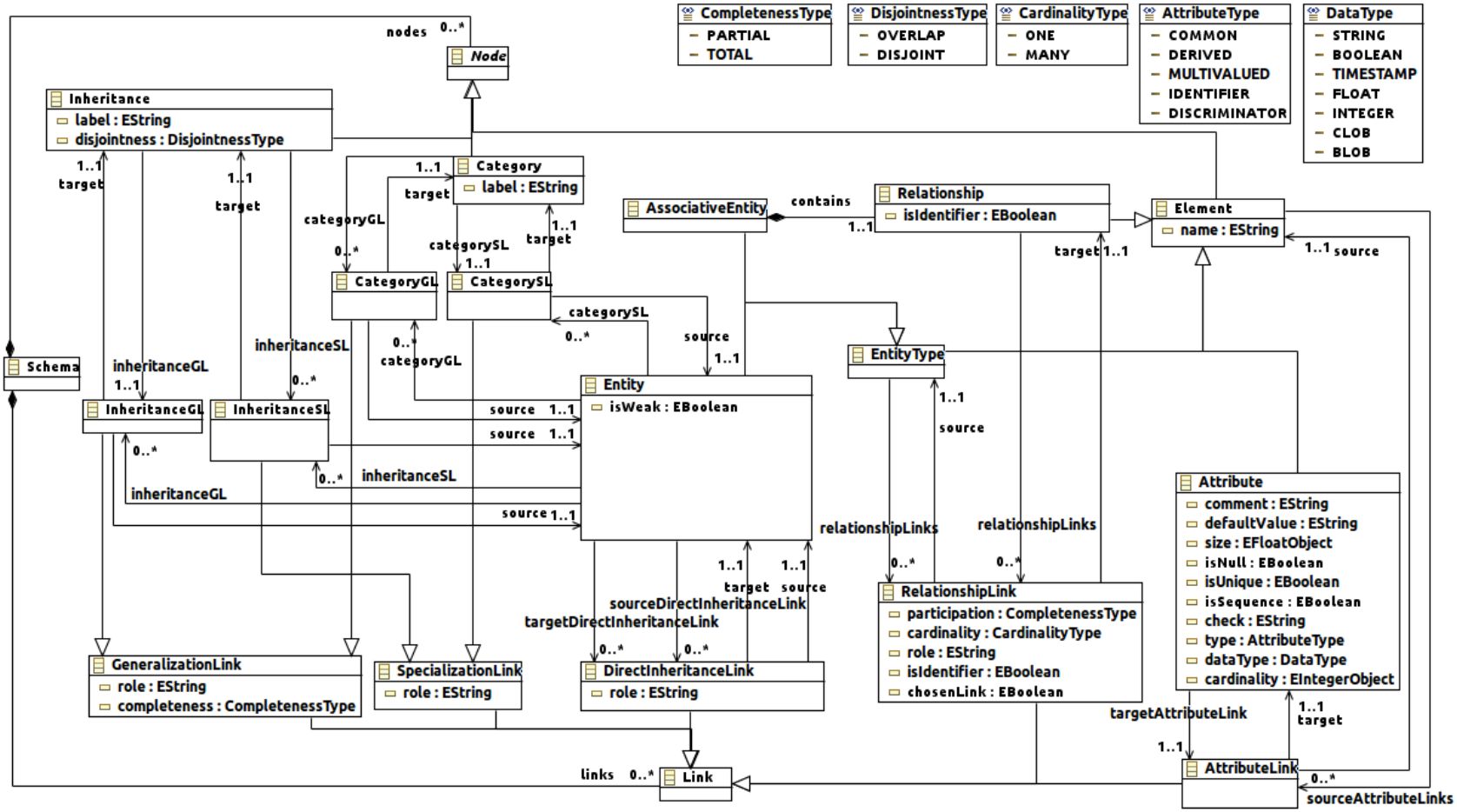
Na Figura 6, apresenta-se o EERMM. Nesta figura um Esquema (Schema) é composto por meta-classes do tipo Nó (Node) e do tipo Ligação (Link). A meta-classe Nó é especializada nas meta-classes: Herança (Inheritance), Categoria (Category) e Elemento (Element), a qual é especializada em Atributo (Attribute), Relacionamento (Relationship) e Entidade Associativa (AssociativeEntity). Sobre as meta-classe de Ligação, pode-se observar que esta tem um Nó como origem (source) e outro Nó como destino (target). Além disso, também se pode observar que a meta-classe Ligação é especializada nas seguintes meta-classes: Ligação de Atributo (AttributeLink), Ligação de Relacionamento (RelationshipLink), Ligação de Herança Direta (DirectInheritanceLink), Ligação de Generalização, a qual é especializada em Ligação de Generalização de Herança (InheritanceGL) e Ligação de Generalização de Categoria (CategoryGL) e, por fim, Ligação de

Especialização, a qual é especializada em Ligação de Especialização de Herança (InheritanceSL) e Ligação de Especialização de Categoria (CategorySL).

O metamodelo EERMM descreve a sintaxe do Modelo EER. No entanto, para que o esquema EER seja semanticamente válido é necessário seguir um conjunto de regras de boa formação (FIDALGO et al., 2013). Apesar das regras para transformação de esquemas EER em esquemas Relacionais serem independentes da validade semântica. A seguir são apresentadas quinze regras de boa formação. Ressalta-se que estas regras são de conhecimento comum e foram descobertas empiricamente pelo autor desta dissertação nos principais livros para BD (ELMASRI; NAVATHE, 2010) (GARCIA-MOLINA; ULLMAN; WIDOM, 2008) (KROENKE; AUER, 2013) (TEOREY et al., 2011), (SILBERSCHATZ; KORTH; SUDARSHAN, 2010) (RAMAKRISHNAN; GEHRKE, 2002) (DATE, 2003) (HEUSER, 1998).

1. Entidade Regular deve ter ao menos um Atributo do tipo identificador;
2. Entidade Fraca não pode ser Sub-Entidade de Herança;
3. Em um Relacionamento Identificador, a ligação da Entidade Forte não pode ter Cardinalidade N. Se a ligação da Entidade Fraca tiver Cardinalidade N, então a Entidade Fraca deve ter ao menos um Atributo Discriminador;
4. Atributo do tipo Discriminador somente pode ser ligado com Entidade Fraca, Relacionamentos Binários N:N, Unários N:N, N-ário, Entidade Associativa conectada com Relacionamento Binário N:N ou Entidade Associativa conectada com Relacionamento N-ário;
5. Em uma Entidade Associativa, o Relacionamento conectado não pode ter a Entidade Associativa como uma de suas ligações;
6. Atributo do tipo Composto não pode passar do nível dois e os Sub-Atributos sempre são do tipo Comum;
7. Atributos devem estar ligados a uma Entidade, Relacionamento ou Atributo;

Figura 6 - Metamodelo EERMM.



Fonte: Adaptado de (FIDALGO et al., 2013)**.**

8. Relacionamento deve estar conectado com no mínimo duas Ligações de Relacionamentos;
9. Entidade Associativa não pode estar ligada a um Relacionamento com Participação Total;
10. Herança com Disjunção (i.e, que não seja Herança Direta) deve ter no mínimo duas Sub-Entidade;
11. Categoria deve ter no mínimo duas Super-Entidades;
12. Sub-Entidade de Herança não pode ter Atributo do tipo Identificador;
13. Relacionamentos não podem estar ligados com Atributos do tipo Identificador;
14. Entidade Associativa não pode conter um Relacionamento Identificador;
15. Entidade Associativa não pode estar associada com um Relacionamento Unário.

A seguir é apresentado o Modelo Relacional.

2.1.5 Modelo Relacional

O Modelo Relacional foi proposto por Codd (CODD, 1970), o qual organiza os dados em termos de tabelas, que consistem em linhas, onde cada linha é descrita por valores de colunas, no qual, em uma linha, cada coluna pode ter somente um valor. Cada tabela armazena dados de um único assunto e as linhas desta tabela são instâncias únicas de um determinado conjunto de dados. Estas colunas com valores únicos constituem a chave primária da tabela. Uma linha de uma tabela pode estar conectada a uma coluna de outra tabela, de modo que o valor referenciado fornece informações necessárias para encontrar a linha apontada naquela tabela. Estas colunas constituem a chave estrangeira da tabela.

Um esquema Relacional pode ser especificado de forma textual (ELMASRI; NAVATHE, 2010). Nesta notação, cada tabela pode ser mostrada como uma linha de nome de colunas. O nome da tabela é escrito antes do

nome das colunas. A chave primária da tabela são as colunas sublinhadas. No Quadro 1, é mostrado a notação textual que expressa um esquema Relacional.

Quadro 1 - Notação textual para o Modelo Relacional.

NOME_TABELA(<u>chavePrimaria</u>, campoObrigatório!, [CampoÚnico], CampoChaveEstrangeira) campoChaveEstrangeira **REF** NOME_TABELA (campoChavePrimária) **GATILHO**("lógica")

Fonte: Adaptado de (ELMASRI; NAVATHE, 2010)**.**

No Quadro 1, *NOME_TABELA* é uma tabela do esquema Relacional, composta pelas seguintes colunas: *<u>chavePrimaria</u>, campoObrigatório!, [CampoÚnico], CampoChaveEstrangeira*. Note que o sublinhado denota que o campo (ou composição deste) forma a Chave Primária da Tabela. A exclamação indica que o campo é de preenchimento obrigatório. Os colchetes ilustram que o campo (ou composição destes) apenas admite valores (ou combinação deste) que sejam únicos. REF denota que um campo (ou combinação destes) forma uma Chave Estrangeira. Por fim, GATILHO informa que é necessário definir um gatilho no SGBD. A seguir, apresenta-se um metamodelo para o Modelo Relacional.

2.1.6 Metamodelo IMM Relacional

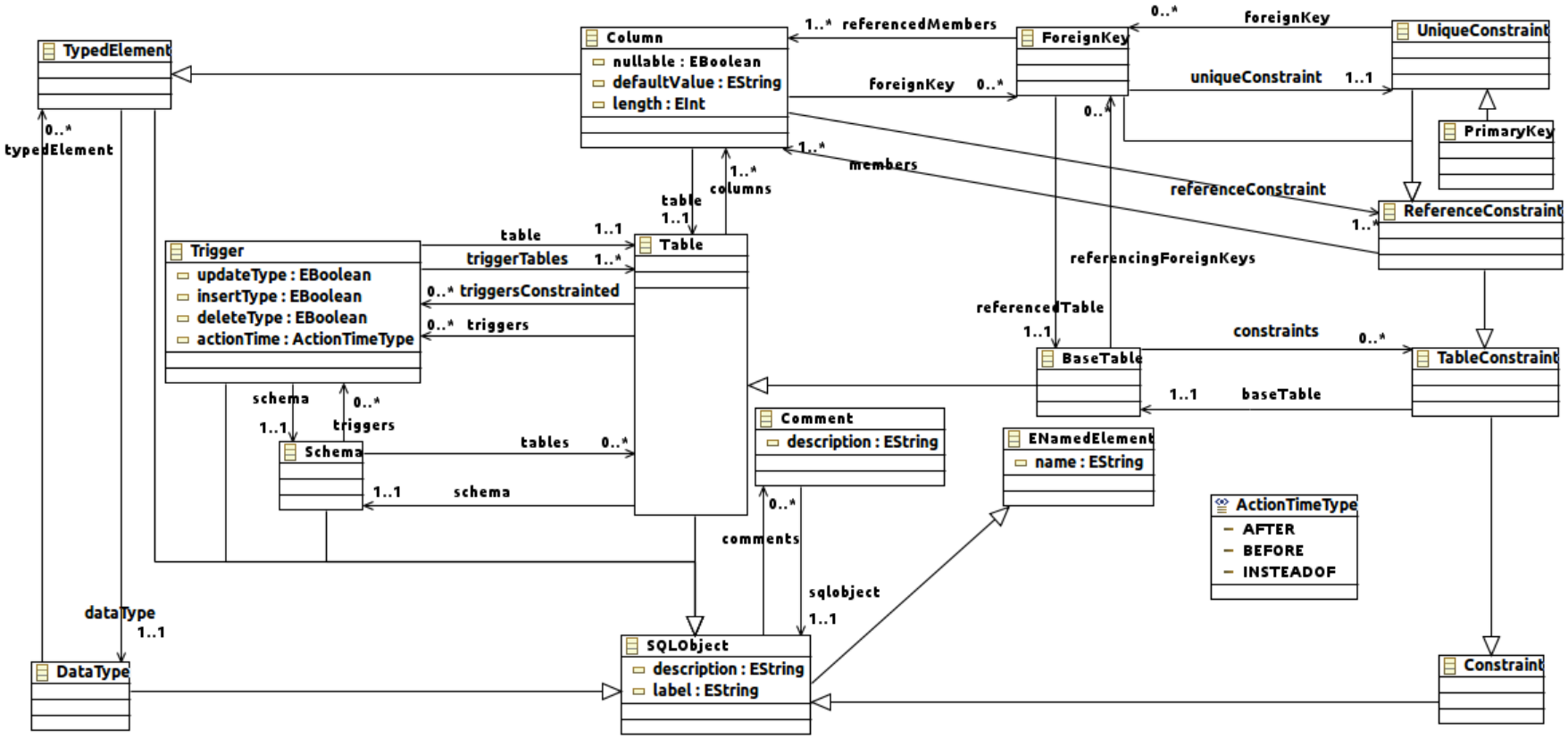
O *Information Management Metamodel* (IMM) (OMG, 2009) é uma especificação que define um metamodelo para o Modelo Relacional. Neste metamodelo são descritos princípios básicos do Modelo Relacional, com base nos conceitos do padrão SQL 2003(ISO, 2003). Uma visão da implementação do metamodelo IMM Relacional é mostrada na Figura 7.

Na Figura 7, apresenta-se o metamodelo IMM Relacional. Neste metamodelo, defini-se que as meta-classes (Schema), (Table), (Column), (Constraint), (UniqueKey), (PrimaryKey), (ForeignKey) e (DataType) correspondem respectivamente aos conceitos de Schema, Tabela, Coluna, Restrição, Restrição de Unicidade, Chave Primária, Chave Estrangeira e tipo de dados do Modelo Relacional. A meta-classe (Table) é uma coleção de uma ou mais (Columns), que podem ter valores em uma ou mais linhas de (Table) e a meta-classe (Column) deve ser definida por exatamente uma instância da meta-classe (DataType). Sobre a meta-classe (Schema), pode-se observar que

este tem uma coleção de (Tables) e (Triggers). Além disso, também se pode observar que a meta-classe (PrimaryKey) é uma (UniqueKey), no qual (UniqueKey) deve ser identificada por instâncias de exatamente uma (Table) e composta por uma ou mais (Columns). A meta-classe (ForeignKey) precisar especificar a (PrimaryKey) que ela aponta. Por fim, a meta-classe (Trigger) representa um fragmento de um programa especifico de uma (Table), que pode ocorre em diferentes eventos (i.e., *update*, *insert* ou *delete*) e tempo (i.e., *before*, *insteof* of ou *after*).

Assim como o EERMM, o metamodelo IMM Relacional descreve a sintaxe do Modelo Relacional, no entanto, na modelagem do esquema Relacional é necessário seguir três regras de boa formação para que este possa ser válido semanticamente (OMG, 2009). Estas regras de boa formação são enumeradas a seguir: 1) uma Tabela pode ter somente uma Chave Primária; 2) uma Coluna não pode existir sem estar relacionada com uma Tabela; e 3) uma Chave Estrangeira deve apontar para uma Chave Primária ou restrição de unicidade. Vale ressaltar que a completude dessas regras está fora do escopo deste trabalho. A seguir são apresentadas as considerações finais do capítulo.

Figura 7 - Metamodelo IMM para Banco de Dados Relacional.



Fonte: Adaptado de (OMG, 2009)**.**

2.2 Considerações Finais

Neste Capítulo, foi discutida a importância da Transformação de Modelos no paradigma MDD, juntamente com seus conceitos básicos. Em seguida, o Modelo EER e o metamodelo EERMM foram abordados. Na sequência, foi pesquisado o Modelo Relacional e o metamodelo IMM Relacional. Estes conceitos são necessários para solução proposta neste trabalho, pois a mesma é feita com base em MDD, utiliza Transformação de Modelos e os metamodelos EERMM e IMM Relacional. No próximo capítulo, são discutidos trabalhos que estão relacionados a esta dissertação.

3 Trabalhos Relacionados

Neste Capítulo são discutidos três trabalhos relacionados à transformação de esquemas EER em esquemas Relacionais (EMBLEY; MOK, 2011) (HAINAUT, 2006) (CUADRA et al., 2002). Os critérios para a escolha destes trabalhos foram: 1) contemplar regras de transformação não ambíguas (i.e., regras apresentadas em linguagem matemática ou algorítmica); 2) oferecer suporte para garantir a semântica da restrição de Participação em Relacionamentos e 3) suportar a garantia da semântica das restrições de Completude e Disjunção em Heranças e Categorias. Portanto, excluiram desta análise trabalhos clássicos como: (ELMASRI; NAVATHE, 2010), (GARCIA-MOLINA; ULLMAN; WIDOM, 2008), (KROENKE; AUER, 2013), (TEOREY et al., 2011), (SILBERSCHATZ; KORTH; SUDARSHAN, 2010), (RAMAKRISHNAN; GEHRKE, 2002), (DATE, 2003) e (HEUSER, 1998), pois estes apresentam as regras de transformação usando linguagem natural. O restante deste Capítulo está organizado da seguinte forma: apresentação das características usadas para avaliar os trabalhos relacionados; análise de cada trabalho relacionado, destacando-se seus pontos fortes e fracos; e, por fim, comparação dos trabalhos de acordo com características previamente definidas.

3.1 Características para avaliação dos trabalhos

Buscando fazer uma análise equitativa dos trabalhos relacionados, a seguir, são apresentados os critérios usados para avaliá-los. Para cada trabalho é avaliado se este cobre (**Sim**), não cobre (**Não**) ou cobre parcialmente (**Parcial**) cada um dos critérios de avaliação. Vale ressaltar que as regras para transformação de Atributos, Entidades e Cardinalidade não são consideradas, pois estas são cobertas por todos os trabalhos citados.

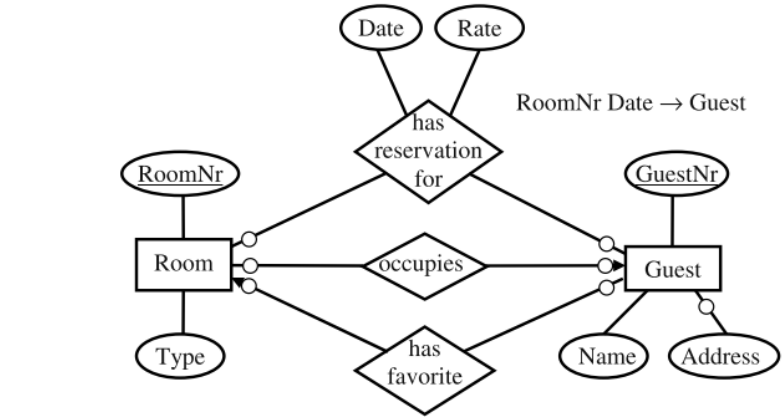
1. **Participação em Relacionamento Binário**;
2. **Participação em Relacionamento N-ário**;
3. **Participação em Relacionamento Unário**;
4. **Participação em Relacionamento Identificador**;
5. **Participação em Entidade Associativa**;

6. **Disjunção em Herança**;
7. **Completude em Herança**;
8. **Completude em Categoria**;
9. **Baseado em MDD.**

3.2 Embley e Mok

O trabalho de Embley e Mok (EMBLEY; MOK, 2011) propõe regras para transformação da maioria dos conceitos do Modelo EER, faltando somente os conceitos de Entidade Associativa e Categoria. Na Figura 8, tem-se um exemplo que mostra um esquema EER simplificado e seu esquema Relacional transformado segundo as regras em questão. Ressalta-se que a notação usada por Embley e Mok tem Participação e Cardinalidade com leitura "*Look Across*".

Figura 8 - Transformação de Relacionamentos com Restrição de Participação por Embley e Mok.



(a) Esquema ER

RoomAndOccupant(RoomNr, Type, GuestNr?)
GuestAndFavoriteRoom(GuestNr, Name, Address?, RoomNr?)
HasReservationFor(RoomNr, Date, GuestNr, Rate)

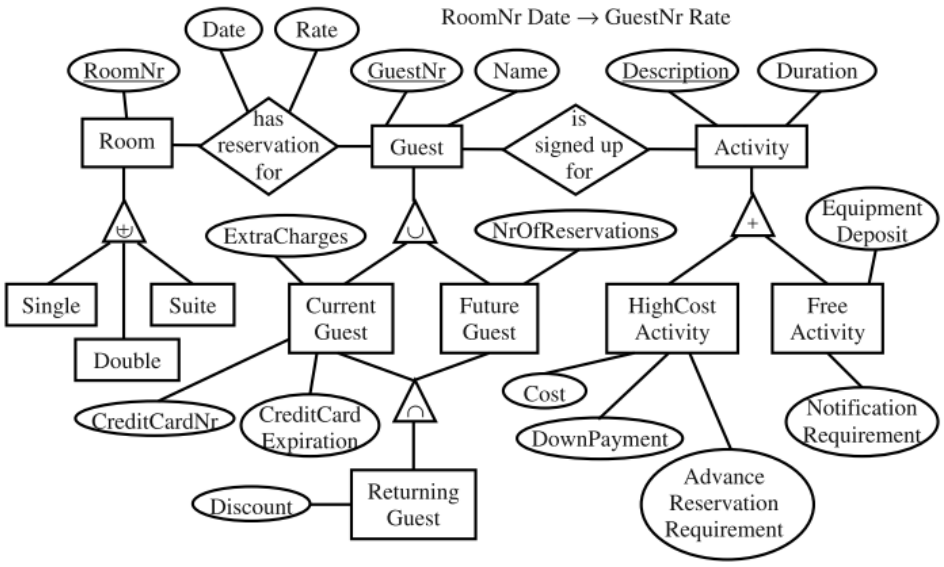
(b) Esquema Relacional

Fonte: Embley e Mok (2011).

Na Figura 8a, os símbolos "▸" e "o" denotam, respectivamente, o lado 1 do Relacionamento e que a Participação é Parcial. Por sua vez, os símbolos "?" e "____" (sublinhado) na Figura 8b indicam, respectivamente, quais Colunas são anuláveis e quais são Chaves Primárias. Posto isto, pode-se observar na Figura 8b que a Participação Parcial é tratada, de forma trivial, informando que a Chave Estrangeira é anulável (e.g., *GestNR* em *RoomAndOccupant*). Contudo, vale ressaltar que as regras de transformação em questão não tratam restrições de Participação não triviais (cf.Figura 2).

Na Figura 9, tem-se um exemplo que mostra um esquema EER com Heranças e seu esquema Relacional transformado segundo as regras de Embley e Mok. Na Figura 9a, os símbolos “⊎”, “+”, “U” e “∩”, denotam, respectivamente, Herança Disjunta Total, Herança Disjunta Parcial, Herança Sobreposta Total e Herança Sobreposta Parcial. Na Figura 9b, pode-se observar que as restrições de Disjunção e Completude de uma Herança não podem ser garantidas pelo SGBD, pois não há a definição de Gatilhos para garanti-las. Uma análise sintetizada sobre o trabalho de Embley e Mok é apresentada no Quadro 2.

Figura 9 - Transformação de Herança com restrição de Completude por Embley e Mok.



(a) Esquema ER

Room(RoomNr, RoomType)
Guest(GuestNr, Name, ExtraCharges?, CreditCardNr?, CreditCardExpiration?, NrOfReservations?, Discount?)
Activity(Description, Duration)
HighCostActivity(Description, Cost, DownPayment, AdvanceReservationRequirement)
FreeActivity(Description, EquipmentDeposit, NotificationRequirement)
HasReservationFor(RoomNr, Date, GuestNr, Rate)
IsSignedUpFor(GuestNr, Description)

(b) Esquema Relacional

Fonte: Embley e Mok (2011).

Quadro 2 - Avaliação do trabalho de Embley e Mok.

Características de Avaliação	
Relacionamento Binário (Participação)	**Parcial**
Relacionamento N-ário (Participação)	**Parcial**
Relacionamento Unário (Participação)	**Parcial**
Relacionamento Identificador (Participação)	**Parcial**
Entidade Associativa (Participação)	**Não**
Herança (Restrição de Disjunção)	**Não**
Herança (Restrição de Completude)	**Não**
Categoria (Restrição de Completude)	**Não**
Baseado em MDD	**Não**
% de Sim	**0%**
% de Não	**55,5%**
% de Parcial	**44,4%**

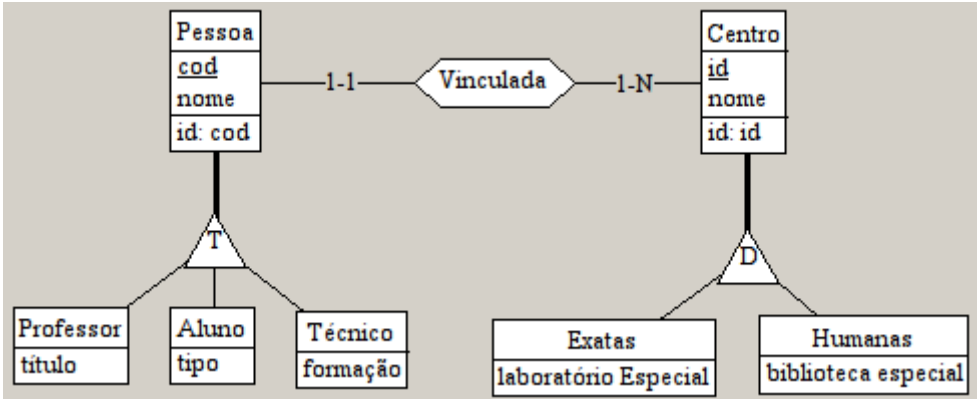
Fonte: Autor (2015).

A avaliação apresentada no Quadro 2 tem como base as seguintes constatações sobre o trabalho: 1) as regras de transformação não tratam os casos de Relacionamentos com Participação não trivial (cf. Figura 2), os quais só podem ser tratados a partir do uso de Gatilhos ou Assertivas, recursos que não são abordados no trabalho; 2) não especifica como transformar Entidades Associativas; 3) as regras para transformar Herança não permitem que o SGBD garanta as restrições Disjunta e Total, pois também exigem a definição de Gatilhos ou Assertivas para serem tratadas pelo SGBD; por fim, 4) não é baseada em MDD (i.e., metamodelos e transformações de modelos).

3.3 Hainaut

O trabalho de Hainaut (HAINAUT, 2006) é fundamentado no paradigma de MDD e as regras de transformação são descritas com auxilio de metamodelo e implementadas na ferramenta DB-MAIN (DBMAIN, 2015). Na Figura 10, ilustra-se um esquema EER e no Quadro 3, o seu respectivo código SQL transformado na ferramenta DB-MAIN. Ressalta-se que a notação usada por Hainaut tem Participação e Cardinalidade com leitura “*Look Here*”.

Figura 10 - Esquema EER para avaliar as regras de transformação por Hainaut.



Fonte: Autor (2015).

Na Figura 10, é mostrado um esquema EER no qual a Entidade *Pessoa* é Super-Entidade na Herança Sobreposta Total, que tem como Sub-Entidades *Professor*, *Aluno* e *Técnico*. Nesta figura também é mostrado que *Centro* é Super-Entidade da Herança Disjunta Parcial, que tem como Sub-Entidades *Exatas* e *Humanas*. Além disso, observa-se o Relacionamento Binário 1:N com Participação não trivial entre as Entidades *Pessoa* e *Centro*.

Quadro 3 - Código SQL gerado a partir das regras de transformação por Hainaut.

```
create table Professor (
     cod numeric(10) not null,
     titulo char(1) not null,
     constraint FKPes_Pro_ID primary key (cod));
create table Aluno (
     cod numeric(10) not null,
     tipo char(1) not null,
     constraint FKPes_Alu_ID primary key (cod));
create table Pessoa (
     cod numeric(10) not null,
     nome char(1) not null,
     id numeric(10) not null,
     Tecnico numeric(10),
     Professor numeric(10),
     Aluno numeric(10),
     constraint ID_Pessoa_ID primary key (cod));
create table Centro (
     id numeric(10) not null,
     nome char(1) not null,
     Humanas numeric(10),
     Exatas numeric(10),
     constraint ID_Centro_ID primary key (id));
create table Tecnico (
     cod numeric(10) not null,
     formacao char(1) not null,
     constraint FKPes_Tec_ID primary key (cod));
```

```
create table Exatas (
     id numeric(10) not null,
     laboratorio_Especial char(1) not null,
     constraint FKCen_Exa_ID primary key (id));
create table Humanas (
     id numeric(10) not null,
     biblioteca_especial char(1) not null,
     constraint FKCen_Hum_ID primary key (id));
-- Constraints Section
alter table Professor add constraint FKPes_Pro_FK
     foreign key (cod)
     references Pessoa;
alter table Aluno add constraint FKPes_Alu_FK
     foreign key (cod)
     references Pessoa;
alter table Pessoa add constraint LSTONE_Pessoa
     check(Tecnico is not null or Aluno is not null or Professor is
not null);
alter table Pessoa add constraint FKVinculada_FK
     foreign key (id)
     references Centro;
alter table Centro add constraint ID_Centro_CHK
     check(exists(select * from Pessoa
                  where Pessoa.id = id));
alter table Centro add constraint EXCL_Centro
     check((Humanas is not null and Exatas is null)
           or (Humanas is null and Exatas is not null)
           or (Humanas is null and Exatas is null));
alter table Tecnico add constraint FKPes_Tec_FK
     foreign key (cod)
     references Pessoa;
alter table Exatas add constraint FKCen_Exa_FK
     foreign key (id)
     references Centro;
alter table Humanas add constraint FKCen_Hum_FK
     foreign key (id)
```

Fonte: Autor (2015).

No Quadro 3, apresenta-se o código SQL-Relacional, gerado na ferramenta DB-MAIN, correspondente ao esquema EER da Figura 10. Como pode ser visto, para tratar a Participação não trivial é utilizado restrições de Cheques na Tabela *Centro*, com objetivo de garantir que um registro de *Centro* somente seja inserido no BD caso este for relacionado com ao menos um registro de *Aluno*. No entanto, esta restrição não é suficiente para garantir esta Participação não trivial, pois é necessário que ao atualizar ou excluir um registro de *Aluno*, todo registro de *Centro* possua ao menos um registro de *Aluno*. Além disso, observa-se que as restrições de Disjunção e Completude das Heranças não podem ser garantidas pelo SGBD, pois as restrições de Cheques definidas não são suficientes para garanti-las. Uma análise sintetizada sobre o trabalho de Hainaut é apresentada no Quadro 4.

Quadro 4 - Avaliação do trabalho de Hainaut.

Características de Avaliação	
Relacionamento Binário (Participação)	**Parcial**
Relacionamento N-ário (Participação)	**Parcial**
Relacionamento Unário (Participação)	**Parcial**
Relacionamento Identificador (Participação)	**Não**
Entidade Associativa (Participação)	**Não**
Herança (Restrição de Disjunção)	**Não**
Herança (Restrição de Completude)	**Não**
Categoria (Restrição de Completude)	**Não**
Baseado em MDD	**Sim**
% de Sim	**11,1%**
% de Não	**55,5%**
% de Parcial	**33,3%**

Fonte: Autor (2015).

A avaliação apresentada no Quadro 4 tem como base as seguintes constatações sobre o trabalho: 1) apesar da definição de restrições de Cheques, as regras de transformação não tratam os casos de Relacionamentos com Participação não trivial; 2) não especifica como transformar Relacionamentos Identificadores e Entidades Associativas; 3) as regras para transformar Herança não permitem que o SGBD garanta as restrições Disjunta e Total, pois a especificação de Cheques não são suficientes para garantir a consistência do BD; por fim, 4) é baseada em MDD (i.e., usa metamodelos e transformação de modelos).

3.4 Cuadra *et al*.

Cuadra *et al.* (CUADRA et al., 2002) propõem regras para transformação de Relacionamentos Binários e N-ários que têm restrição de Participação não trivial. Nos Relacionamentos Binários são especificados Gatilhos para os seguintes casos: 1) N:N com pelo menos uma das ligações como Total; 2) 1:1 com as duas ligações como Totais; e 3) 1:N com a ligação do lado 1 como Total. No Quadro 5 é mostrado um exemplo de Gatilho para especificar as restrições de Participação Total ao inserir, atualizar ou excluir um registro em um Relacionamento 1:N.

Quadro 5 - Gatilho para garantir a Participação total ao inserir, atualizar ou excluir.

```
CREATE TRIGGER INSERTION_1N_(1,N)_E1 BEFORE INSERT ON E1 FOR EACH ROW
BEGIN
ASK_PK(E2,:VK2);
IF NOT EXISTS(SELECT * FROM E2 WHERE K2=:VK2)
THEN
   ASK_REST (E2,:VREST_ATTRIBUTES_E2);
   INSERT INTO E2 (K2, REST_ATTRIBUTES_E2, K1)
   VALUES (:VK2, :VREST_ATTRIBUTES_E2,:NEW.K1)
ELSE
   UPDATE E2
   SET K1=:NEW.K1
   WHERE K2=:VK2
END;

CREATE TRIGGER DELETION_1N_(1,N)_E2 BEFORE DELETE ON E2 FOR EACH ROW
BEGIN
IF :OLD.K2 IN (SELECT K2 FROM E2 WHERE K1 IN
   (SELECT K1 FROM E2 GROUP BY K1 HAVING
   COUNT(*)=1))
THEN ROLLBACK
END;

CREATE TRIGGER UPDATE_1N_(1,N)_E2 BEFORE UPDATE ON E2 FOR EACH ROW BEGIN
IF :OLD.K1<>:NEW.K1 AND :OLD.K2 IN
   (SELECT K2 FROM E2 WHERE K1 IN
   (SELECT K1 FROM E2 GROUP BY K1 HAVING
   COUNT(*)=1))
THEN ROLLBACK
END;
```

Fonte: Cuadra *et al* (2002).

No Quadro 5, apresentam-se Gatilhos para garantir a Participação Total ao inserir (i.e, *INSERTION_1N_*), atualizar (i.e., *UPDATE_1N_*) ou excluir (i.e., *DELETION_1N_*) um registro na Tabela criada a partir da Entidade do lado Total do Relacionamento. No Gatilho para garantir a inserção (i.e., *INSERTION_1N_*), são utilizadas as funções "*ASK_PK(tabela, chave primária)"* que obtém a Chave Primária de uma Tabela e "ASK_REST(tabela, rest_atributos)" que obtém todos os Atributos de uma Tabela, com exceção dos que compõe a Chave Primária. No momento em que este Gatilho é disparado, é verificada se existe algum registro que contenha a Chave Estrangeira para esta Tabela. Caso não exista, é inserido um novo registro na Tabela que contem a Chave Estrangeira, sendo que a Chave Estrangeira deste registro

aponta para o registro que disparou o Gatilho. A análise feita sobre o trabalho de Cuadra *et al* é resumida no Quadro 6.

Quadro 6 - Avaliação da proposta de Cuadra *et al*.

Características de Avaliação	
Relacionamento Binário (Participação)	**Sim**
Relacionamento N-ário (Participação)	**Sim**
Relacionamento Unário (Participação)	**Sim**
Relacionamento Identificador (Participação)	**Não**
Entidade Associativa (Participação)	**Não**
Herança (Restrição de Disjunção)	**Não**
Herança (Restrição de Completude)	**Não**
Categoria (Restrição de Completude)	**Não**
Baseado em MDD	**Não**
% de Sim	**33,3%**
% de Não	**77,7%**
% de Parcial	**0%**

Fonte: Autor (2015).

A avaliação apresentada no Quadro 6 tem como base as seguintes constatações sobre o trabalho: 1) a transformação da Participação é especificada corretamente para os casos de Relacionamento Binário, Unário e N-ário; 2) não especifica como transformar Relacionamento Identificador, Entidades Associativas e Herança ou Categoria; e 3) não é baseada em MDD.

3.5 Análise consolidada

Uma análise consolidada dos trabalhos relacionados é apresentada no Quadro 7. Sobre este quadro vale destacar os seguintes pontos:

- Todos os trabalhos abordam restrições de Participação em Relacionamentos Binários, N-ários e Unários. No entanto, apenas o trabalho de Cuadra *et al*. especifica Gatilhos para garantir as restrições de Participação não triviais corretamente;
- Apenas o trabalho de Embley e Mok considera parcialmente as restrições de Participação em Relacionamento Identificador;
- Nenhum trabalho aborda Participação em Entidade Associativa, Disjunção em Herança ou Completude em Herança e Categoria;
- Somente o trabalho de Hainaut é baseado em MDD.

Quadro 7 - Análise consolidada entre os trabalhos avaliados.

Trabalhos / Características de Avaliação	Embley e Mok	Hainaut	Cuadra *et al.*	% Sim	% Não	% Parcial
Relacionamento Binário (Participação)	Parcial	Parcial	Sim	33,3%	0%	66,6%
Relacionamento N-ário (Participação)	Parcial	Parcial	Sim	33,3%	0%	66,6%
Relacionamento Unário (Participação)	Parcial	Parcial	Sim	33,3%	0%	66,6%
Relacionamento Identificador (Participação)	Parcial	Não	Não	0%	66,6%	33,3%
Entidade Associativa (Participação)	Não	Não	Não	0%	100%	0%
Herança (Restrição de Disjunção)	Não	Não	Não	0%	100%	0%
Herança (Restrição de Completude)	Não	Não	Não	0%	100%	0%
Categoria (Restrição de Completude)	Não	Não	Não	0%	100%	0%
Baseado em MDD	Não	Sim	Não	33,3%	66,6%	0%

Fonte: Autor (2015).

No próximo capítulo são apresentadas as regras para transformações de esquemas EER em esquemas Relacionais propostas nesta dissertação.

4 Regras para Transformação Automática de Esquemas EER em Esquemas Relacionais e Código SQL

A transformação de um esquema EER em esquema Relacional e este em código SQL não é uma tarefa trivial, principalmente quando o objetivo é automatizar a execução das regras de transformação. Visando mostrar como este objetivo é realizado, este capítulo está organizado da seguinte forma: inicialmente são discutidas as dependências estruturais entre os construtores EER. Na sequência, o Catálogo de regras para transformar um esquema EER em um esquema Relacional e este em código SQL é apresentado. Por fim, um algoritmo que permite executar as regras de transformação em uma ordem correta é discutido.

4.1 Dependências estruturais entre os construtores EER

Conforme exposto na Seção 1.2, a transformação de um esquema EER em um esquema Relacional não é um processo que pode ser automatizado de forma sequencial, pois existem construções que têm dependências estruturais que devem ser executadas primeiro que outras. Assim, a ordem de execução das regras de transformações é definida a partir de uma análise dessas dependências no esquema EER modelado.

Com base no contexto acima, uma matriz que mostra as combinações permitidas e as dependências estruturais entre os principais construtores do Modelo EER é apresentada no Quadro 8. Ressalta-se que: 1) os diferentes tipos de Atributos não entraram no Quadro 8, pois isto iria poluí-lo sem acrescentar informação útil, uma vez que Atributos podem ocorrer em qualquer tipo de Entidade e Relacionamento e sua criação sempre depende destes; 2) a leitura do Quadro 8 deve ser feita linha a linha, obedecendo a direção da esquerda para a direita e combinando o construtor da linha com cada construtor da coluna. Isto é, verificando se o construtor listado na linha pode ter

dependência estrutural com o construtor exibido na coluna; 3) as células que não estão preenchidas indicam que não há dependência entre os construtores; 4) as células marcadas com o símbolo "⃠" expressam que a combinação entre os construtores não é permitida pelo metamodelo EERMM (cf. Seção 2.1.4); 5) as células preenchidas com "↥" apontam a direção da dependência estrutural entre os construtores; e 7) os papeis "For", "Fra", "Sup", "Sub", "Con" e "Ass" são abreviações que correspondem, respectivamente, aos seguintes conceitos: "Entidade **For**te", "Entidade **Fra**ca", "**Sup**er-Entidade", "**Sub**-Entidade", "Entidade Associativa **Con**tendo um Relacionamento" e "Entidade Associativa se **Ass**ociando com um Relacionamento".

Quadro 8 - Matriz de permissões e dependências estruturais do Modelo EER.

	Construtor	Entidade	Rel. Binário	Rel. N-ário	Rel. Unário	Rel. Ident.	Herança	Categoria	Entidade Associativa
1	Entidade	⃠				For: / Fra: ↥(1)	Sup: / Sub: ↥(2)	Sup: / Sub: ↥(3)	⃠
2	Rel. Binário	↥	⃠	⃠	⃠	⃠	⃠	⃠	↥
3	Rel. N-ário	↥	⃠	⃠	⃠	⃠	⃠	⃠	↥
4	Rel. Unário	↥	⃠	⃠	⃠	⃠	⃠	⃠	⃠
5	Rel. Identificador	For: / Fra: ↥(1)	⃠	⃠	⃠	⃠	⃠	⃠	For: / Fra: ⃠
6	Herança	Sup: / Sub: ↥(2)	⃠	⃠	⃠	⃠	⃠	⃠	⃠
7	Categoria	Sup: / Sub: ↥(3)	⃠	⃠	⃠	⃠	⃠	⃠	⃠
8	Entidade Associativa	⃠	Con: ↥ / Ass:	Con: ↥ / Ass:	⃠	For: / Fra: ⃠	⃠	⃠	⃠

Fonte: Autor (2015).

Na linha 1 do Quadro 8 mostra-se que o construtor Entidade não tem dependência estrutural com a maioria dos tipos de Relacionamento, pois, inicialmente, a transformação de Entidades em Tabelas não exige a criação das restrições de Chaves Estrangeiras (estas são criadas na transformação dos Relacionamentos). A exceção do caso anterior é o Relacionamento Identificador. Neste caso, quando a Entidade é Fraca, esta depende do identificador que é obtido via o Relacionamento Identificador com a Entidade Forte. Contudo, quando a Entidade é Forte, esta não depende de nenhum

identificador para ser transformada em uma Tabela. Seguindo este raciocínio, em uma Herança ou Categoria, uma Sub-Entidade pode depender do identificador da sua Super-Entidade. Porém, uma Super-Entidade não depende de nenhum identificador para ser transformada em uma Tabela. Ressalta-se que a transformação de Herança múltipla está fora do escopo deste trabalho.

Nas linhas 2, 3 e 4 do Quadro 8 observa-se que a transformação dos Relacionamentos Binários, N-ários e Unários podem depender do identificador das Entidades (incluindo Entidades Associativas) que participam dos Relacionamentos. A exceção a este caso ocorre em Entidade Associativa com Relacionamento Unário, o que é proibido pelo metamodelo EERMM.

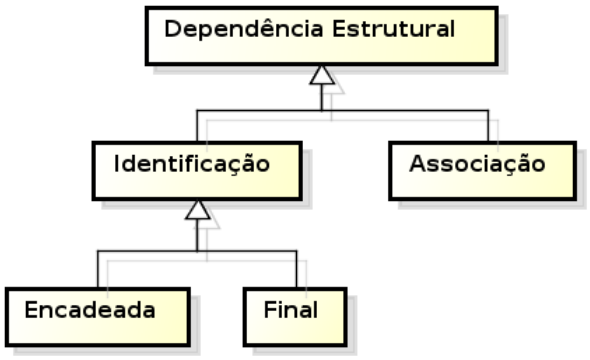
Nas linhas 5, 6 e 7 do Quadro 8 os construtores Relacionamento Identificador, Herança e Categoria seguem o mesmo raciocínio usado com o construtor Entidade da linha 1 e os números delimitados por parênteses indicam os casos que seguem o mesmo raciocínio. Ressalta-se que, na combinação entre os construtores Relacionamento Identificador e Entidade Associativa, o metamodelo EERMM não permite que uma Entidade Associativa seja Fraca. Além disso, quando esta for Forte, seguindo o mesmo raciocínio usado com o construtor Entidade, não há nenhuma dependência estrutural para transformá-la em uma Tabela.

Na linha 8 do Quadro 8 tem-se que uma Entidade Associativa não depende de nenhum tipo de Relacionamento para ser transformada em uma Tabela. Contudo, observa-se que uma Entidade Associativa depende que o Relacionamento que esta contém já tenha sido transformado em uma Tabela, pois, na transformação EER em Relacional, a Entidade Associativa e o Relacionamento que esta contém, correspondem à mesma Tabela. Por fim, a combinação entre os construtores Entidade Associativa e Relacionamento Identificador segue o mesmo raciocínio apresentado na linha 5.

Em resumo, como pode ser observado no Quadro 8, existem dez combinações distintas que têm dependências, portanto, necessitam que a ordem de execução das transformações dos construtores envolvidos seja analisada e definida. As demais combinações não exigem esta análise, pois

não têm dependência entre os construtores envolvidos ou são proibidas pelo metamodelo EERMM. Ou seja, quando permitidas, podem ser executadas em qualquer ordem. De modo a definir uma ordem para a execução correta das regras de transformações, uma taxonomia de dependências estruturais é apresentada no Diagrama de Classes na Figura 11.

Figura 11 - Taxonomia de Dependências Estruturais.



Fonte: Autor (2015).

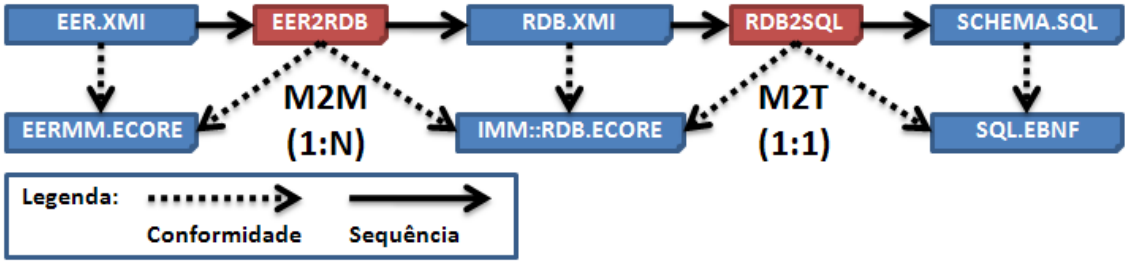
Na Figura 11, uma dependência estrutural pode ser de três tipos: 1) identificação e encadeada, 2) identificação e final ou 3) apenas de associação. No primeiro caso – identificação e encadeada, tem-se que a Chave Primária da Tabela a ser transforma pode ter dependências sucessivas de Chaves Primárias de outras Tabelas. Por exemplo, se a Tabela a ser transformada corresponder a uma Sub-Entidade de outra Sub-Entidade ou corresponder a uma Entidade Fraca cuja sua Entidade Forte também é uma Entidade Fraca. O segundo caso – identificação e final - contradiz o primeiro, pois informa que não existe a possibilidade de ocorrer dependências sucessivas de Chaves Primárias. Por exemplo, uma Tabela transformada a partir de um Relacionamento N:N ou de um Relacionamento N-ário. Por fim, no terceiro caso – associação - denota-se que uma Tabela transformada recebe uma Chave Estrangeira. Por exemplo, uma Tabela transformada é atualizada para receber uma Chave Estrangeira referente à transformação de um Relacionamento 1:N ou 1:1. Ressalta-se que toda dependência de identificação gera uma Chave Primária que também é Chave Estrangeira. Contudo, em uma dependência de associação, uma Chave Estrangeira nunca pode ser Chave Primária.

Com base no que é discutido no Quadro 8, a Seção 4.2 apresenta o Catálogo de Regras de Transformação proposto neste trabalho. Por sua vez, com base na Figura 11, a Seção 4.3 apresenta um algoritmo para executar as regras de transformações.

4.2 Catálogo de Regras para Transformação

O Catálogo apresentado nesta Seção utiliza os metamodelos EERMM e IMM Relacional (cf. Seção 2.1.2) como base para especificar as regras que permitem, de forma automática, transformar esquemas EER em esquemas Relacionais. Na Figura 12, ilustra-se o uso destas regras em um processo de transformação MDD. Ressalta-se que o EERMM foi adaptado nesta dissertação, com objetivo de auxiliar a transformação do esquema EER para esquema Relacional. Nesta adaptação foi adicionada a propriedade "*chosenLink*" na meta-classe (RelationshipLink) para impedir que existam casos ambíguos de transformação (e.g., Relacionamento Binário 1:1 com Participações Totais). Pois, com essa informação, é possível que o projetista especifique com detalhes o comportamento da transformação.

Figura 12 - Aplicação do Catálogo no processo de transformações MDD.



Fonte: Autor (2015).

Na Figura 12, a transformação ocorre em dois passos, sendo o primeiro uma transformação modelo para modelo (M2M) - do esquema EER para o esquema Relacional - e o segundo, uma transformação modelo para texto (M2T) - do esquema Relacional para código SQL-Relacional. Destaca-se que: 1) a transformação M2M não é direta, pois a transformação de um elemento do EERMM pode corresponder a mais de um elemento do IMM (i.e., a transformação M2M é 1:N); e 2) a transformação M2T é direta (i.e., esta transformação é 1:1). Posto isto, o Catálogo de regras de transformação

especificado neste trabalho é utilizado para implementar a transformação M2M (EER2RDB), as quais geram um esquema lógico Relacional que é mapeado diretamente (RDB2SQL) para um esquema físico em SQL. Neste contexto, considerando o construtor Atributo e os oito construtores do Quadro 8, tem-se nove regras de transformação, as quais são apresentadas no Quadro 9.

Quadro 9 - Algoritmos que especificam as regras de transformação.

	Regra	Algoritmo	Descrição
1	RTA	ATA1	Transformar Atributos do tipo comum ou derivado.
		ATA2	Transformar Atributos do tipo identificador ou discriminador.
		ATA3	Transformar Atributos do tipo multivalorado.
2	RTER	ATER	Transformar Entidades Regulares.
3	RTRB	ATRB11	Transformar Relacionamentos Binários 1:1.
		ATRB1N	Transformar Relacionamentos Binários 1:N.
		ATRBNN	Transformar Relacionamentos Binários N:N.
4	RTRU	ATRU11	Transformar Relacionamentos Unários 1:1.
		ATRU1N	Transformar Relacionamentos Unários 1:N.
		ATRUNN	Transformar Relacionamentos Unários N:N.
5	RTRN	ATRN	Transformar Relacionamentos N-ários.
6	RTRI	ATRI	Transformar Relacionamentos Identificadores.
7	RTH	ATHD	Transformar Heranças Diretas.
		ATH	Transformar Heranças.
8	RTC	ATC	Transformar Categorias.
9	RTEA	ATEARB	Transformar Entidades Associativas Contendo Relacionamentos Binários.
		ATEARN	Transformar Entidades Associativas Contendo Relacionamentos N-ários.

Fonte: Autor (2015).

É importante destacar que as regras de transformação apresentadas no Quadro 9 são definidas de modo a garantir que: 1) um esquema EER não pode ser transformado em dois ou mais esquemas Relacionais diferentes; 2) para ser fiel ao esquema EER modelado, apesar de ser possível transformar/fundir duas ou mais Entidades em uma única Tabela (e.g., a transformação/fusão das Entidades de uma Herança em uma única Tabela), as regras de transformação são especificadas de modo a gerarem uma Tabela para cada Entidade ou Relacionamento; e 3) todas as restrições de Cardinalidade e Participação de Relacionamento, bem como de Disjunção e Completude de Herança e Categoria são refletidas no esquema Relacional. Apesar desta última garantia ser tão importante quanto às anteriores, ressalta-se que poucos trabalhos (cf.

Capítulo 3) cobrem os caso não triviais de restrições de Participação em Relacionamento e restrições de Disjunção e Completude em Herança e Categoria.

Para facilitar a especificação e explicação dos algoritmos referentes às regras de transformação, utilizam-se as funções *isTable(e:EERMM::EntityType, r:IMM::relationalSchema):boolean* e *getTable (e:EERMM::EntityType, r:IMM::relationalSchema):table*. Estas funções têm como entrada dois parâmetros: 1) um construtor do tipo Entidade E (i.e., Regular, Fraca, Associativa, Super ou Sub); e 2) a atual versão do esquema Relacional que está sendo transformado. A função *isTable* é utilizada para verificar se o construtor do tipo Entidade já foi transformado em uma Tabela. Por sua vez, a função *getTable* é usada para obter o objeto Tabela que corresponde ao construtor do tipo Entidade passado como parâmetro. Ressalta-se que a função *getTable* é utilizada apenas quando o retorno da função *isTable* for verdadeiro. Na seção a seguir, é apresentada a regra para transformação de Atributos.

4.2.1 Regra para transformação de Atributos

A Regra para Transformação de Atributos (RTA) é modularizada em três algoritmos: ATA1 (cf. Algoritmo 1), ATA2 (cf. Algoritmo 2) e ATA3 (cf. Algoritmo 3). O ATA1 transforma Atributos Comuns ou Derivados, o ATA2, Atributos Identificadores ou Discriminadores e o ATA3, Atributos Multivalorados. Ressalta-se que a RTA poderia ser especificada em um único algoritmo. Contudo, para facilitar a sua compreensão, optou-se por modularizá-la nos algoritmos em questão. Além disso, é importante destacar que a função *attribute2Column(attribute:EERMM::Attribute):IMM::Column* recebe um Atributo e transforma-o em uma Coluna. Isto é, mapeia o nome, o tipo do dado, o tamanho, a unicidade, a obrigatoriedade e o valor padrão do Atributo para as respectivas propriedades de uma Coluna, a qual é retornada pela função. O pseudocódigo do algoritmo ATA1 é mostrado a seguir.

Algoritmo 1: ATA1 - Transformação de Atributo Comum ou Derivado

```
Input:      element: EERMM::Element, table: IMM::BaseTable;
Output:     table: IMM::BaseTable;
Variables:attributeLink : EERMM::AttributeLink;
            attribute, compositeAttribute:EERMM::Attribute;
            column :IMM::Column;

1.  for each attributeLink ∈ element.sourceAttributeLinks[] do
2.    attribute ← attributeLink.targetAttribute;
3.    if ( attribute.type = "COMMON" or  attribute.type = "DERIVED" ) then
4.      if (attribute.sourceAttributeLinks[].size() = 0 )          /* atributo simples */
5.        column ← attribute2Column(attribute);
6.        table.columns[].add(column);
7.      else                                                       /* atributo composto */
8.        for each attributeLink ∈ attribute.sourceAttributeLinks[] do
9.          compositeAttribute ← attributeLink.targetAttribute;
10.         column ← attribute2Column(compositeAttribute);
11.         table.columns[].add(column);
12.       end for
13.     end if
14.     if (attribute.type = "DERIVED") then
15.       table.triggers.add("Trigger a ser definida pelo administrador do BD");
16.     end if
17.   end if
18. end for
19. return table;
```

No Algoritmo 1, têm-se como entrada os objetos *element* e *table* que correspondem a um *Element* do EERMM e a uma *BaseTable* do IMM, respectivamente. Por sua vez, a saída deste algoritmo corresponde ao objeto *table* com seus Atributos Comuns e Derivados criados. O corpo do algoritmo inicia percorrendo cada Atributo de *element*. Para cada Atributo percorrido, faz-se a verificação se o Atributo é Comum ou Derivado (cf. linha 3) e, na sequência, é feita a verificação se o Atributo é Simples ou Composto. Se o Atributo for Simples, este é diretamente transformado em uma Coluna, que é adicionada a coleção de Colunas de *table* (cf. linhas de 4 a 6). No caso de um Atributo ser Composto, para cada Sub-Atributo, este é transformado em uma Coluna, que também é adicionada a coleção de Coluna de *table* (cf. linhas de 7 a 12). Na sequência, se o Atributo for Derivado, cria-se um Gatilho cujo corpo deve ser definido pelo administrador do banco de dados (cf. linhas 14 a 16). Por fim, *table* é retornada (cf. linha 19). Na sequência, o pseudocódigo do algoritmo ATA2 é discutido.

Algoritmo 2: ATA2 - Transformação de Atributo Identificador ou Discriminador

```
Input:      element: EERMM::Element, table: IMM::BaseTable;
Output:     table: IMM::BaseTable;
Variables: attributeLink: EERMM::AttributeLink;
           attribute, compositeAttribute: EERMM::Attribute;
           column: IMM::Column;
           primaryKey: IMM::PrimaryKey;

1.  for each attributeLink ∈ element.sourceAttributeLinks[] do
2.    attribute ← attributeLink.targetAttribute;
3.    if ( attribute.type = "IDENTIFIER" or  attribute.type = "DISCRIMINATOR" ) then
4.      if (attribute.sourceAttributeLinks[].size() = 0 ) then      /* atributo simples */
5.        column ← attribute2Column(attribute);
6.        table.columns[].add(column);
7.        primaryKey.members[].add(column);
8.      else                                                        /* atributo composto */
9.        for each attributeLink ∈ attribute.sourceAttributeLinks[] do
10.         compositeAttribute ← attributeLink.targetAttribute;
11.         column ← attribute2Column(compositeAttribute);
12.         table.columns[].add(column);
13.         primaryKey.members[].add(column);
14.       end for
15.     end if
16.   end if
17. end for
18. table.constraints[].add(primaryKey);
19. return table;
```

A diferença do algoritmo ATA2 para o ATA1 dá-se pela adição do Atributo Identificador ou Discriminador à coleção de membros que formam a Chave Primária (cf. linhas 7 e 13), que é adicionada à coleção de restrições de *table* (cf. linha 18). O pseudocódigo do algoritmo ATA3 é visto a seguir.

Algoritmo 3: ATA3 - Transformação de Atributo Multivalorado

```
Input:      element: EERMM::Element, table: IMM::BaseTable;
Output:     table: IMM::BaseTable;
Variables: attributeLink : EERMM::AttributeLink;
           attribute, compositeAttribute :EERMM::Attribute;
           column :IMM::Column;
           primaryKey : IMM::PrimaryKey;
           foreignKey : IMM::ForeignKey;
           multAttTable : IMM::BaseTable;

1.  for each attributeLink ∈ element.sourceAttributeLinks[] do
2.    attribute ← attributeLink.targetAttribute;
3.    if(attribute.type = "MULTIVALUED") then
4.      multAttTable.name ← attribute.name;
5.      multAttTable.columns[].add( table.primaryKey.columns[]);
6.      foreignKey.referencedTable ← table;
7.      foreignKey.uniqueConstraint ← table.primaryKey;
8.      foreignKey.referencedMembers[].add(table.primaryKey.columns[]);
9.      foreignKey.setNullable(false);                  /* as colunas não são anuláveis */
10.     if(attribute.sourceAttributeLinks[].size() = 0) then      /*atributo simples */
11.       column ← attribute2Column(attribute);
```

```
12.          multAttTable.columns[].add(column);
13.          primaryKey.members[].add(column);
14.        else                                          /* atributo composto */
15.          for each attributeLink ∈ attribute.sourceAttributeLinks[] do
16.            compositeAttribute ← attributeLink.targetAttribute;
17.            column ← attribute2Column(compositeAttribute);
18.            multAttTable.columns[].add(column);
19.            primaryKey.members[].add(column);
20.          end for
21.        end if
22.      end if
23.      if(attribute.isUnique() = false ) then
24.        primaryKey.members[].add(table.primaryKey.columns[]);
25.      end if
26.    end for
27.    multAttTable.constraints[].add(foreignKey);
28.    multAttTable.constraints[].add(primaryKey);
29.    return multAttTable;
```

A diferença do algoritmo ATA3 para os demais está no fato que, para cada Atributo Multivalorado de *element* é criada uma Tabela (i.e., *multAttTable*), que, inicialmente, tem como Colunas a coleção de Colunas que formam a Chave Primária da Tabela origem (cf. linha 5). Depois disso, na Tabela *multAttTable*, cria-se a Chave Estrangeira para a Tabela origem (cf. linhas de 6 a 9) e testa-se se o Atributo é Simples ou Composto. No caso do Atributo ser Simples, faz-se a transformação do Atributo Multivalorado em uma Coluna, a qual irá compor a coleção de Colunas da Tabela *multAttTable,* bem como formar parte da sua Chave Primária (cf. linhas de 10 a 13). Se o Atributo for Composto, para cada Sub-Atributo faz-se a sua transformação em uma Coluna, a qual é adicionada à coleção de Colunas da Tabela *multAttTable* e fará parte da sua Chave Primária (cf. linhas 15 a 20). No caso do Atributo Multivalorado não ser único, a Chave Primária da Tabela origem é adicionada à Chave Primária da Tabela *multAttTable* (cf. linhas 23 a 25). Por fim, as Chaves Primária e Estrangeira são adicionadas como restrições de *multAttTable* (cf. 27 e 28). A seguir, a regra para transformar Entidades Regulares e exemplos dos usos dos Algoritmos ATA1, ATA2 e ATA3 são apresentados.

4.2.2 Regra para transformação de Entidade Regular

A Regra para Transformação de Entidade Regular (RTER) é especificada usando o Algoritmo ATER (cf. Algoritmo 4), que faz uso dos algoritmos ATA1, ATA2 e ATA3.

Algoritmo 4: ATER -Transformação de Entidade Regular

```
Input:      eerSchema: EERMM::Schema;
Output:     rdbSchema: IMM::Schema;
Variables: table : IMM::BaseTable;
           entity : EERMM::Entity;
           node : EERMM::Node;
1.  for each node ∈ eerSchema.nodes[] do
2.    if (node.isTypeOf(EERMM::Entity) = true) then
3.      entity ← node;
4.      if(entity.targetDirectInheritanceLink[].size() = 0 and
          entity.inheritanceSL[].size() = 0 and
          entity.categorySL[].size() = 0
          entity.isWeak = false) then
5.        table.name ← entity.name;
6.        table ← ATA1(entity, table);
7.        table ← ATA2(entity, table);
8.        rdbSchema.add(table);
9.        rdbSchema.add( ATA3(entity, table) );
10.     end if
11.   end if
12. end for
13. return rdbSchema;
```

No Algoritmo 4, tem-se como entrada um esquema EER e como saída um esquema Relacional. A execução do algoritmo se inicia percorrendo cada Nó do Esquema EER modelado, testando e capturando aqueles que correspondem a uma Entidade (cf. linhas de 1 a 3). Na sequência, para cada Entidade, é verificado se esta não é Sub-Entidade de Herança, Sub-Entidade de Categoria ou Entidade Fraca (cf. linha 4). Se o resultado for verdadeiro, cria-se uma Tabela com o mesmo nome da Entidade (cf. linha 5), executam-se as regras RTA1 e RTA2 (cf. linhas 6 e 7), adiciona-se a Tabela criada ao esquema Relacional (cf. linha 8) e executa-se a regra RTA3, a qual retorna uma Tabela que é adicionada ao Esquema Relacional (cf. linha 9). Por fim, o Esquema Relacional é retornado.

Na Figura 13, apresenta-se um exemplo de utilização da RTER. Nesta figura o esquema EER na parte “a” é transformado para o esquema Relacional exibido na parte “b”. Vale ressaltar, apesar de ser possível na notação do EER, que o Atributo *multA1* foi definido como único e o Atributo *multA2* como não único.

Figura 13 - Transformação de Entidade Regular.

Fonte: Autor (2015).

4.2.3 Regra para transformação de Relacionamento Binário

A Regra para Transformação de Relacionamento Binário (RTRB) é modularizada em três algoritmos: ATRB11 (cf. Algoritmo 5), ATRB1N (cf. Algoritmo 6) e ATRBNN (cf. Algoritmo 7), os quais transformam Relacionamentos Binários com Cardinalidade 1:1, 1:N e M:N, respectivamente. Vale destacar que a especificação destes algoritmos requer a definição de Gatilhos para garantir os casos não triviais de Participação Total. Além disso, é importante destacar que a função *getLink(arg:String):EERMM::RelationshipLink* recebe um argumento (e.g., o valor da cardinalidade ou da participação) e retorna a Ligação de Relacionamento que atende ao valor passado como argumento da função.

Algoritmo 5: ATRB11 - Transformação de Relacionamento Binário 1:1

```
Input:      eerSchema : EERMM::Schema, rdbSchema: IMM::Schema;
Output:     rdbSchema: IMM::Schema;
Variables: tableA, tableB : IMM::BaseTable;
           relationship : EERMM::Relationship;
           relationshipLinkA, relationshipLinkB : EERMM::RelationshipLink;
           foreignKey : IMM::ForeignKey;
           uniqueConstraint : IMM::UniqueConstraint;
           node : EERMM::Node;

1.  for each node ∈ eerSchema.nodes[] do
2.    if (node.isTypeOf(EERMM::RelationShip) = true) then
3.      relationship ← node;
4.      if(relationship.relationshipsLinks[].size() = 2 and
          relationship.associativeEntity = null and
          relationship.isIdentifier = false )  then
5.        if (relationship.relationshipsLinks[].getCountOfTotalLinks() = 1)
6.          relationshipLinkA ← relationship.relationshipsLinks[].getLink("PARTIAL");
7.          relationshipLinkB← relationship.relationshipsLinks[].getLink("TOTAL");
8.        else
9.          relationshipLinkA ← relationship.relationshipsLinks[].getLink("ChosenLinkFalse");
10.         relationshipLinkB ← relationship.relationshipsLinks[].getLink("ChosenLinkTrue");
11.       end if
```

```
12.        if(isTable(relationshipLinkA.source, rdbSchema) = true and
             isTable(relationshipLinkB.source, rdbSchema) = true) then
13.          tableA ← getTable(relationshipLinkA.source, rdbSchema);
14.          tableB ← getTable(relationshipLinkB.source, rdbSchema);
15.          if( tableA <>tableB and
               relationshipLinkA.cardinality = "1" and
               relationshipLinkB.cardinality = "1") then
16.            foreignKey.referencedTable ← tableA;
17.            foreignKey.uniqueConstraint ← tableA.primaryKey;
18.            foreignKey.members[].add(tableA.primaryKey.columns[]);
19.            uniqueConstraint.members[].add(tableA.primaryKey.columns[]);
20.            tableB.constraints[].add(uniqueConstraint);
21.            tableB.columns[].add(tableA.primaryKey.columns[]);
22.            if(relationshipLinkA.participation = "PARTIAL" and
                 relationshipLinkB.participation = "PARTIAL") then
23.              foreignKey.setNullable(true);              /* as colunas são anuláveis */
24.            else
25.              if(relationshipLinkB.participation = "TOTAL") then
26.                foreignKey.setNullable(false);           /* as colunas não são anuláveis */
27.              end if
28.              if(relationshipLinkA.participation = "TOTAL") then
29.                tableA.triggers.add("Após inserir um registro 'a' em tabela A, garantir que existe
                     um registro b em tabela B referenciando o registro 'a' ");
30.                  tableB.triggers.add("Após remover ou atualizar um registro b em tabela B, garantir
                       que todo registro da tabela A é referenciado por pelo menos um
                       registro da tabela B ");
31.              end if
32.            end if
33.            tableB.constraints[].add(foreignKey);
34.            tableB ← ATA1(relationship, tableB);
35.            rdbSchema.add( ATA3(relationship, tableB) );
36.            rdbSchema.update(tableA);
37.            rdbSchema.update(tableB);
38.          end if
39.        end if
40.      end if
41.    end if
42. end for
43. return rdbSchema;
```

O Algoritmo 5 recebe como entrada um esquema EER a ser transformado e um esquema Relacional correspondente ao que já foi transformado. Por sua vez, a saída deste algoritmo corresponde ao esquema Relacional com os Relacionamentos Binários 1:1 que possuem suas dependências estruturais satisfeitas transformados. A execução do algoritmo se inicia percorrendo cada Nó do esquema EER modelado, testando e ajustando-o para que este corresponda a um Relacionamento (cf. linhas de 1 a 3). Na sequência, para cada Relacionamento, é verificado se este é binário, não está contido em uma Entidade Associativa e não é Identificador (cf. linha 4). Depois disso, é verificado se a quantidade de ligações com Participação

Total do Relacionamento é igual a um. Caso o resultado seja verdadeiro, convenciona-se que as Entidades correspondentes as ligações marcadas como "*PARTIAL"* e "*TOTAL*" são, respectivamente, manipuladas pelos objetos *relationshipLinkA* e *relationshipLinkB* (cf. linhas de 5 a 7). Caso o resultado seja falso, convenciona-se que as Entidades das ligações marcadas como "*ChosenLinkFalse" e "ChosenLinkTrue"* são, respectivamente, manipuladas pelos objetos *relationshipLinkA* e *relationshipLinkB* (cf. linhas de 9 a 11). Destaca-se que esta convenção permite definir que a Tabela correspondente ao objeto *relationshipLinkB* terá uma Chave Estrangeira, não nula, para a Tabela que corresponde ao objeto relationshipLinkA. Em seguida, é testado se as dependências estruturais do Relacionamento foram satisfeitas utilizando a função *isTable* (cf. linha 12). Caso o resultado seja falso, a transformação do Relacionamento é cancelada e inicia-se a transformação do próximo Relacionamento. Do contrario, a transformação do Relacionamento continua. Na sequência, é testado se o Relacionamento não é Unário e se a Cardinalidade é 1:1 (cf. linha 15). Caso este teste seja verdadeiro, com base na convenção acima, define-se: 1) qual é a Tabela referenciada pela Chave Estrangeira (cf. linha 16); 2) que a Chave Estrangeira aponta para a Chave Primária da Tabela referenciada (cf. linha 17); 3) quais são as Colunas da Chave Primária que formam a Chave estrangeira (cf. linha 18); e 4) que as Colunas usadas como Chave Estrangeira são únicas (cf. linhas 19 e 20) e fazem parte do conjunto de Colunas da Tabela que a contém (cf. linha 21).

Os passos seguintes do Algoritmo 5 tratam da Participação (Total ou Parcial) do Relacionamento. Neste sentido, se a Participação é Parcial-Parcial, a Chave Estrangeira pode ter valores nulos (cf. linhas 22 e 23). Caso contrário, se a Participação da Entidade correspondente a *relationshipLinkB* for Total, a Chave Estrangeira não pode admitir valores nulos (cf. linhas 25 e 26). Ademais, se a Participação da Entidade correspondente a *relationshipLinkA* for Total (cf. linha 28), devem-se criar dois Gatilhos: um na Tabela referenciada pela Chave Estrangeira (cf. linha 29) e o outro na Tabela que contém a Chave Estrangeira (cf. linha 30). Ressalta-se que estes gatilhos foram descritos em linguagem natural, pois não existe uma linguagem padronizada para isto. Contudo, um exemplo da implementação destes gatilhos em código SQL-Relacional

(PostgreSQL 9.3) pode ser visto no Quadro 10. Por fim, a Chave Estrangeira é adicionada como uma das restrições de sua Tabela (cf. linha 33), os possíveis atributos do Relacionamento são transformados (cf. linhas 34 e 35), as mudanças ocorridas nas Tabelas correspondentes a *relationshipLinkA* e *relationshipLinkB* são atualizadas no Esquema Relacional (cf. linhas 36 e 37), e este é retornado (cf. linha 43).

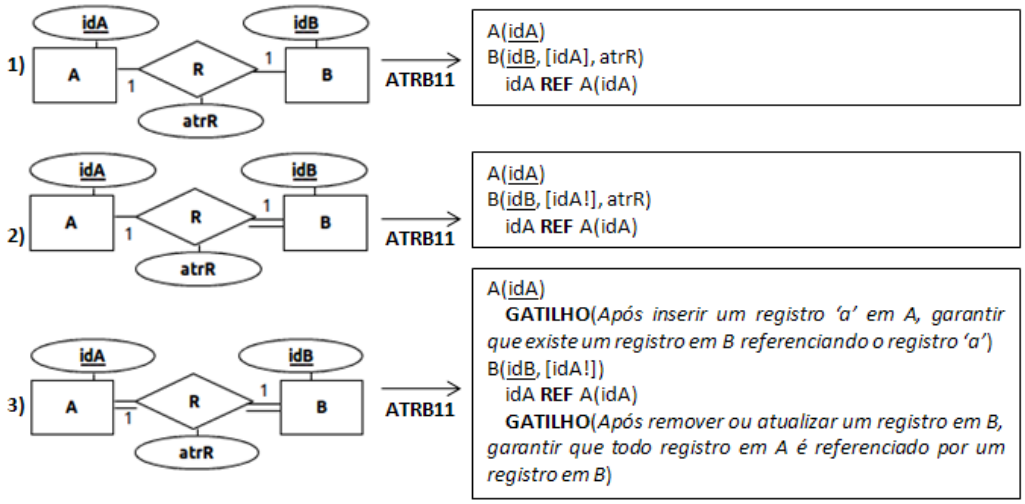
Quadro 10 - Exemplo de Gatilho

```
/* O gatilho TRG_R1_A garante a participação total de A em R1 depois
de inserir um registro em A. Lança uma exceção quando não houver
registro da FK de A em B */
CREATE FUNCTION FUNCTION_TRG_R1_A()
RETURNS TRIGGER AS $BODY$
    DECLARE countA INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countA
        FROM B
        WHERE B.R1_A_idA = NEW.idA;
        IF countA = 0 THEN
            RAISE EXCEPTION 'TRG_R1_A';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_A AFTER INSERT ON A INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R1_A();
```

Fonte: Autor (2015).

Na Figura 14, o Algoritmo ATRB11 é aplicado sobre os três possíveis casos de Cardinalidade e Participação em um Relacionamento Binário 1:1. A seguir, apresenta-se o algoritmo para transformar Relacionamentos Binários 1:N.

Figura 14 - Transformação de Relacionamento Binário 1:1.



Fonte: Autor (2015).

Algoritmo 6: ATRB1N – Transformação de Relacionamento Binário 1:N

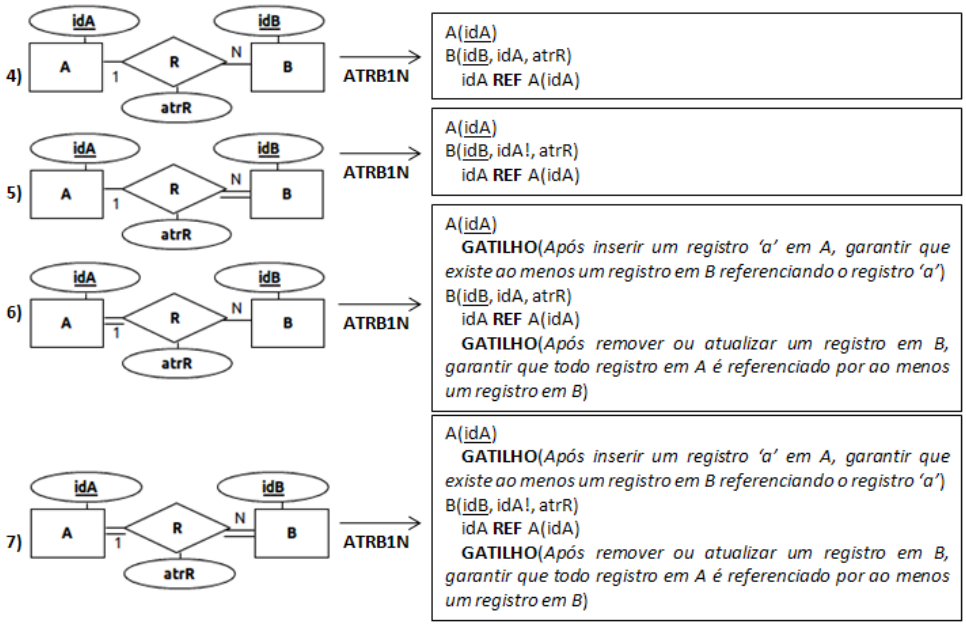
Input: eerSchema : EERMM::Schema, rdbSchema: IMM::Schema;
Output: rdbSchema: IMM::Schema;
Variables: tableA, tableB : IMM::BaseTable;
relationship : EERMM::Relationship;
relationshipLinkA, relationshipLinkB : EERMM::RelationshipLink;
foreignKey : IMM::ForeignKey;
node : EERMM::Node;

```
1.  for each node ∈ eerSchema.nodes[] do
2.    if (node.isTypeOf(EERMM::RelationShip) = true) then
3.      relationship ← node;
4.      if(relationship.relationshipsLinks[].size() = 2 and
          relationship.associativeEntity = null and
          relationship.isIdentifier = false )  then
5.        relationshipLinkA ← relationship.relationshipsLinks[].getLink("1");
6.        relationshipLinkB ← relationship.relationshipsLinks[].getLink("N");
7.        if(isTable(relationshipLinkA.source, rdbSchema) = true and
            isTable(relationshipLinkB.source, rdbSchema) = true) then
8.          tableA ← getTable(relationshipLinkA.source, rdbSchema);
9.          tableB ← getTable(relationshipLinkB.source, rdbSchema);
10.         if(tableA <> tableB and
              relationshipLinkA.cardinality = "1" and
              relationshipLinkB.cardinality = "N" ) then
11.           foreignKey.referencedTable ← tableA;
12.           foreignKey.uniqueConstraint ← tableA.primaryKey;
13.           foreignKey.members[].add( tableA.primaryKey.columns[]);
14.           tableB.columns[].add(tableA.primaryKey.columns[]);
15.           if(relationshipLinkB.participation = "PARTIAL") then
16.             foreignKey.setNullable(true);          /* as colunas são anuláveis */
17.           else
18.             foreignKey.setNullable(false);         /* as colunas não são anuláveis */
19.           end if
20.           if(relationshipLinkA.participation = "TOTAL") then
21.            tableA.triggers.add(" Após inserir um registro 'a' em tabela A, garantir que existe ao
                                    menos um registro em tabela B referenciando o registro 'a' ");
```

```
22.                 tableB.triggers.add(“ Após remover ou atualizar um registro em tabela B, garantir
                                        que todo registro em tabela A é referenciado por ao menos um
                                        registro em tabela B”);
23.             end if
24.             tableB.constraints[].add(foreignKey);
25.             tableB ← ATA1(relationship, tableB);
26.             rdbSchema.add( ATA3(relationship, tableB) );
27.             rdbSchema.update(tableB);
28.             rdbSchema.update(tableA);
29.           end if
30.         end if
31.       end if
32. end for
33. return rdbSchema;
```

A diferença do Algoritmo 6 para o Algoritmo 5 está no fato de que, no Algoritmo 6, convenciona-se que as Entidades correspondentes às ligações marcadas como “*1*” e “*N*” são, respectivamente, manipuladas pelos objetos *relationshipLinkA* e *relationshipLinkB* (cf. linhas de 5 e 6). Por sua vez, com base na convenção acima, define-se que a Chave Estrangeira aponta para a Tabela criada a partir da Entidade correspondente a *relationshipLinkA*. Vale ressaltar que não é adicionada restrição de unicidade as Colunas criadas a partir da Chave estrangeira. Na Figura 15, o Algoritmo ATRB1N é aplicado sobre os quatro possíveis casos de Cardinalidade e Participação em um Relacionamento Binário 1:N. A seguir, apresenta-se o algoritmo para transformar Relacionamentos Binários N:N.

Figura 15 - Transformação de Relacionamento Binário 1:N.



Fonte: Autor (2015).

Algoritmo 7: ATRBNN - Transformação de Relacionamento Binário N:N

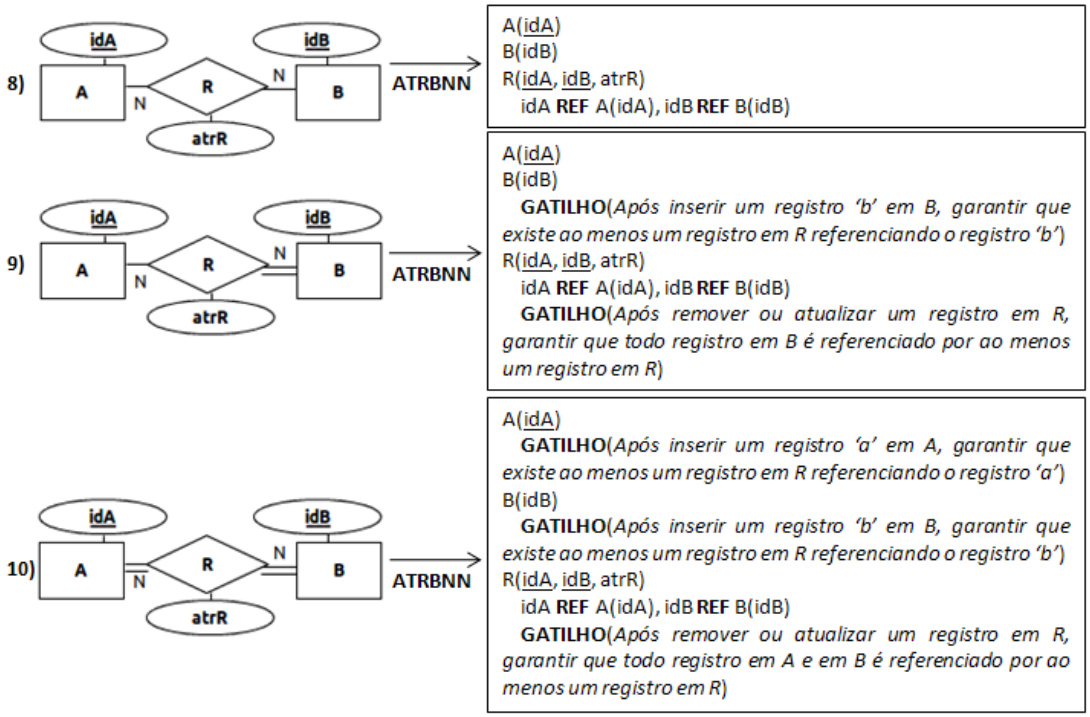
```
Input:      eerSchema : EERMM::Schema, rdbSchema: IMM::Schema;
Output:     rdbSchema: IMM::Schema;
Variables: tableA, tableB, tableNN : IMM::BaseTable;
           relationship : EERMM::Relationship;
           relationshipLinkA, relationshipLinkB : EERMM::RelationshipLink;
           foreignKeyA, foreignKeyB : IMM::ForeignKey;
           node : EERMM::Node;
           primaryKey : IMM::PrimaryKey;

1.  for each node ∈ eerSchema.nodes[] do
2.    if (node.isTypeOf(EERMM::RelationShip) = true) then
3.      relationship ← node;
4.      if(relationship.relationshipsLinks[].size() = 2 and
          relationship.associativeEntity = null and
          relationship.isIdentifier = false )  then
5.        relationshipLinkA ← relationship.relationshipsLinks[1];
6.        relationshipLinkB ← relationship.relationshipsLinks[2];
7.        if(isTable(relationshipLinkA.source, rdbSchema) = true and
            isTable(relationshipLinkB.source, rdbSchema) = true) then
8.          tableA ← getTable(relationshipLinkA.source, rdbSchema);
9.          tableB ← getTable(relationshipLinkB.source, rdbSchema);
10.         if(tableA <> tableB and
              relationshipLinkA.cardinality = "N" and
              relationshipLinkB.cardinality = "N" ) then
11.           tableNN.name ← relationship.name;
12.           foreignKeyA.referencedTable ←tableA;
13.           foreignKeyA.uniqueConstraint ← tableA.primaryKey;
14.           foreignKeyA.members[].add(tableA.primaryKey.columns[]);
15.           tableNN.columns[].add(tableA.primaryKey.columns[]);
16.           tableNN.constraints[].add(foreignKeyA);
17.           foreignKeyB.referencedTable ←tableB;
18.           foreignKeyB.uniqueConstraint ← tableB.primaryKey;
19.           foreignKeyB.members[].add(tableB.primaryKey.columns[]);
20.           tableNN.columns[].add(tableB.primaryKey.columns[]);
21.           tableNN.constraints[].add(foreignKeyB);
22.           primaryKey.members[].add(tableA.primaryKey.columns[]);
23.           primaryKey.members[].add(tableB.primaryKey.columns[]);
24.           tableNN.constraints[].add(primaryKey);
25.           if( relationshipLinkA.participation = "TOTAL" ) then
26.             tableA.triggers.add("Após inserir um registro 'a' em tabela A, garantir que existe ao
                                     menos um registro em tabela NN referenciando o registro 'a' ");
27.             tableNN.triggers.add("Após remover ou atualizar um registro em tabela NN, garantir
                                      que todo registro em tabela A é referenciado por ao menos um
                                      registro em tabela NN");
28.           end if
29.           if( relationshipLinkB.participation = "TOTAL" ) then
30.             tableB.triggers.add("Após inserir um registro 'b' em tabela B, garantir que existe ao
                                     menos um registro em tabela NN referenciando o registro 'b' ");
31.             tableNN.triggers.add("Após remover ou atualizar um registro em tabela NN, garantir
                                      que todo registro em tabela B é referenciado por ao menos um
                                      registro em tabela NN");
32.           end if
33.           tableNN ← ATA1(relationship, tableNN);
34.           tableNN ← ATA2(relationship, tableNN);
35.           rdbSchema.add(tableNN);
36.           rdbSchema.add( ATA3(relationship, tableNN) );
```

```
37.                 rdbSchema.update(tableB);
38.                 rdbSchema.update(tableA);
39.               end if
40.             end if
41.           end if
42.         end if
43. end for
44. return rdbSchema;
```

A diferença do Algoritmo 7 para o Algoritmo 6 está no fato de que, no Algoritmo 7, não existem critérios para convencionar quais ligações do Relacionamento são manipuladas pelos objetos *relationshipLinkA* e *relationshipLinkB* (cf. linhas de 5 a 6). Na sequência, é testado se o Relacionamento não é Unário e se a Cardinalidade é N:N (cf. linha 10). Caso este teste seja verdadeiro, cria-se uma Tabela com o mesmo nome do Relacionamento (cf. linha 11) e com base na convenção acima, define-se: 1) uma Chave Estrangeira para a Tabela criada a partir do Elemento corresponde a *relationshipLinkA* (cf. linhas 12 a 14), que é adicionada como uma das restrições da Tabela correspondente ao Relacionamento (cf. linhas 15 e 16); 2) uma Chave Estrangeira para a Tabela criada a partir do Elemento corresponde a *relationshipLinkB* (cf. linhas 17a 19), que é adicionada como uma das restrições da Tabela correspondente ao Relacionamento (cf. linhas 20 e 21); e 3) uma Chave Primária, no qual são membros as Colunas criadas para as Chaves Estrangeiras (cf. 22 a 23), que é adicionada como uma das restrições da Tabela correspondente ao Relacionamento (cf. linha 24). Ademais, se a Participação de uma ligação do Relacionamento for Total (cf. linhas 25 e 29), criam-se dois Gatilhos: um na Tabela criada a partir do Relacionamento e o outro na Tabela criada a partir do Elemento correspondente à ligação do Relacionamento. Por fim, os possíveis atributos do Relacionamento são transformados (cf. linhas 34 e 35), as mudanças ocorridas nas Tabelas correspondentes a *relationshipLinkA* e *relationshipLinkB* são atualizadas no esquema Relacional (cf. linhas 33, 34 e 36), a Tabela criada é adiciona ao esquema Relacional (cf. linha 35), as Tabelas criadas a partir dos Elementos correspondentes a *relationshipLinkA* e *relationshipLinkB* são atualizadas no esquema Relacional (cf. linhas 37 e 38) e este é retornado (cf. linha 44). Na Figura 16, o Algoritmo ATRBNN é aplicado sobre os três possíveis casos de

Cardinalidade e Participação em um Relacionamento Binário N:N. A seguir, apresenta-se a regra para transformação de Relacionamentos Unários.

Figura 16 - Transformação de Relacionamento Binário N:N.



Fonte: Autor (2015).

4.2.4 Regras para transformação de Relacionamento Unário

A Regra para Transformação de Relacionamento Unário (RTRU) é modularizada em três algoritmos: ATRU11 (cf. Algoritmo 8), ATRU1N (cf. Algoritmo 9) e ATRUNN (cf. Algoritmo 10), os quais transformam Relacionamentos Unários com Cardinalidade 1:1, 1:N e M:N, respectivamente. Vale destacar que duas combinações de Cardinalidade e Participação não são válidas para Relacionamentos Unários, a saber: 1) quando a Cardinalidade for 1:1 e existir uma Participação Total e outra Parcial; e 2) quando a Cardinalidade for 1:N e a do lado 1 tiver Participação Total (DULLEA; SONG; LAMPROU, 2003).

Algoritmo 8: ATRU11 - Transformação de Relacionamento Unário 1:1

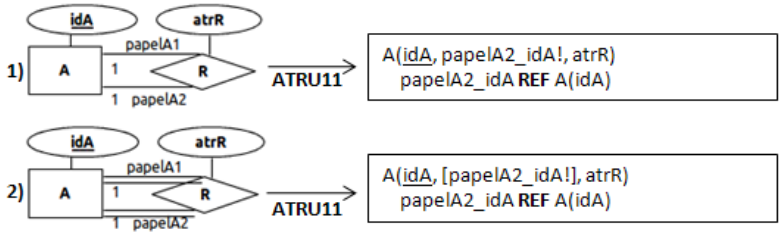
```
Input:      eerSchema : EERMM::Schema, rdbSchema: IMM::Schema;
Output:     rdbSchema: IMM::Schema;
Variables: tableA1, tableA2 : IMM::BaseTable;
           relationship : EERMM::Relationship;
           relationshipLinkA1, relationshipLinkA2 : EERMM::RelationshipLink;
           foreignKey : IMM::ForeignKey;
```

```
            uniqueConstraint : IMM::UniqueConstraint;
            node : EERMM::Node;

1.  for each node ∈ eerSchema.nodes[] do
2.    if (node.isTypeOf(EERMM::RelationShip) = true) then
3.      relationship ← node;
4.      if(relationship.relationshipsLinks[].size() = 2 and
          relationship.associativeEntity = null and
          relationship.isIdentifier = false )  then
5.        relationshipLinkA1 ← relationship.relationshipsLinks[].getLink("ChosenLinkFalse");
6.        relationshipLinkA2 ← relationship.relationshipsLinks[].getLink("ChosenLinkTrue");
7.        if(isTable(relationshipLinkA1.source, rdbSchema) = true and
            isTable(relationshipLinkA2.source, rdbSchema) = true) then
8.          tableA1 ← getTable(relationshipLinkA1.source, rdbSchema);
9.          tableA2 ← getTable(relationshipLinkA2.source, rdbSchema);
10.         if( tableA1 = tableA2 and
              relationshipLinkA1.cardinality = "1" and
              relationshipLinkA2.cardinality = "1") then
11.           foreignKey.referencedTable ← tableA1;
12.           foreignKey.uniqueConstraint ← tableA1.primaryKey;
13.           foreignKey.members[].add(relationshipLinkA1.role + tableA1.primaryKey.columns[]);
14.           if(relationshipLinkA1.participation = "PARTIAL") then
15.             foreignKey.setNullable(true);                    /* as colunas são anuláveis */
16.           else
17.             foreignKey.setNullable(false);                   /* as colunas não são anuláveis */
18.           end if
19.           uniqueConstraint.members[].add(
                relationshipLinkA1.role + tableA1.primaryKey.columns[]);
20.           tableA1.constraints[].add(foreignKey);
21.           tableA1.constraints[].add(uniqueConstraint);
22.           tableA1.columns[].add(relationshipLinkA1.role + tableA1.primaryKey.columns[]);
23.           tableA1 ← ATA1(relationship, tableA1);
24.           rdbSchema.add( ATA3(relationship, tableA1) );
25.           rdbSchema.update(tableA1);
26.         end if
27.       end if
28.     end if
29.   end if
30. end for
31. return rdbSchema;
```

O Algoritmo 8 segue o mesmo raciocínio utilizado no Algoritmo 5, com a diferença que os Elementos correspondentes às ligações marcadas como "*chosenLinkFalse"* e "*chosenLinktrue*" são, respectivamente, manipuladas pelos objetos *relationshipLinkA* e *relationshipLinkB* (cf. linhas de 5 e 6). Além disso, o algoritmo testa se o Relacionamento é Unário (cf. linha 10), neste caso, a Chave Estrangeira criada na Tabela correspondente a *relationshipLinkB* aponta para si mesma (cf. linha 11) e as Colunas da Chave Estrangeira e da restrição de unicidade são criadas com um prefixo corresponde ao Papel da ligação do Relacionamento (cf. linhas 13 e 19). Na Figura 17, o Algoritmo ATRU11 é

aplicado sobre os dois possíveis casos de Cardinalidade e Participação em um Relacionamento Unário 1:1. A seguir, apresenta-se o algoritmo para transformar Relacionamentos Unários 1:N.

Figura 17 - Transformação de Relacionamento Unário 1:1.



Fonte: Autor (2015).

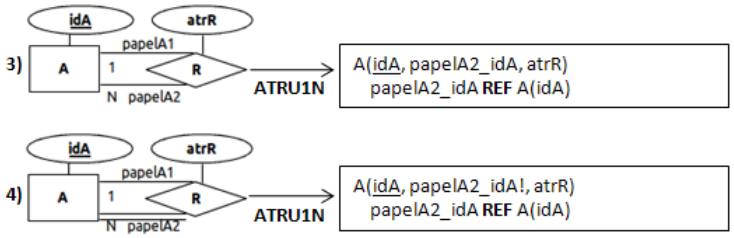
Algoritmo 9: ATRU1N - Transformação de Relacionamento Unário 1:N

```
Input:      eerSchema : EERMM::Schema, rdbSchema: IMM::Schema;
Output:     rdbSchema: IMM::Schema;
Variables: tableA1, tableA2 : IMM::BaseTable;
           relationship : EERMM::Relationship;
           relationshipLinkA1, relationshipLinkA2 : EERMM::RelationshipLink;
           foreignKey : IMM::ForeignKey;
           node : EERMM::Node;

1.  for each node ∈ eerSchema.nodes[] do
2.    if (node.isTypeOf(EERMM::RelationShip) = true) then
3.      relationship ← node;
4.      if(relationship.relationshipsLinks[].size() = 2 and
          relationship.associativeEntity = null and
          relationship.isIdentifier = false )  then
5.        relationshipLinkA1 ← relationship.relationshipsLinks[].getLink("1");
6.        relationshipLinkA2 ← relationship.relationshipsLinks[].getLink("N");
7.        if(isTable(relationshipLinkA1.source, rdbSchema) = true and
            isTable(relationshipLinkA2.source, rdbSchema) = true) then
8.          tableA1 ← getTable(relationshipLinkA1.source, rdbSchema);
9.          tableA2 ← getTable(relationshipLinkA2.source, rdbSchema);
10.         if( tableA1 = tableA2 and
              relationshipLinkA1.cardinality = "1" and
              relationshipLinkA2.cardinality = "N") then
11.           foreignKey.referencedTable ← tableA1;
12.           foreignKey.uniqueConstraint ← tableA1.primaryKey;
13.           foreignKey.members[].add(relationshipLinkA1.role + tableA1.primaryKey.columns[]);
14.           if(relationshipLinkA1.participation = "PARTIAL") then
15.             foreignKey.setNullable(true);                    /* as colunas são anuláveis */
16.           else
17.             foreignKey.setNullable(false);                   /* as colunas não são anuláveis */
18.           end if
19.           tableA1.constraints[].add(foreignKey);
20.           tableA1.columns[].add(relationshipLinkA1.role + tableA1.primaryKey.columns[]);
21.           tableA1 ← ATA1(relationship, tableA1);
22.           rdbSchema.add( ATA3(relationship, tableA1) );
23.           rdbSchema.update(tableA1);
24.         end if
25.       end if
26.     end if
```

```
27.    end if
28.  end for
29.  return rdbSchema;
```

A diferença do Algoritmo 9 para o Algoritmo 8 está no fato de que, no Algoritmo 9, os Elementos correspondentes às ligações com Cardinalidade "*1*" e "*N*" são, manipuladas, respectivamente, pelos objetos *relationshipLinkA* e *relationshipLinkB* (cf. linhas de 5 e 6). Por sua vez, com base na convenção acima, define-se que a Chave Estrangeira aponta para a Tabela criada a partir do Elemento correspondente ao *relationshipLinkA* (cf. linha 11). Vale ressaltar que não é adicionada restrição de unicidade às Colunas criadas a partir da Chave estrangeira. Na Figura 18, o Algoritmo ATRU1N é aplicado sobre os dois possíveis casos de Cardinalidade e Participação em um Relacionamento Unário 1:N. A seguir, apresenta-se o algoritmo para transformar Relacionamentos Unários N:N.

Figura 18 - Transformação de Relacionamento Unário 1:N.



Fonte: Autor (2015).

Algoritmo 10: ATRUNN – Transformação de Relacionamento Unário N:N

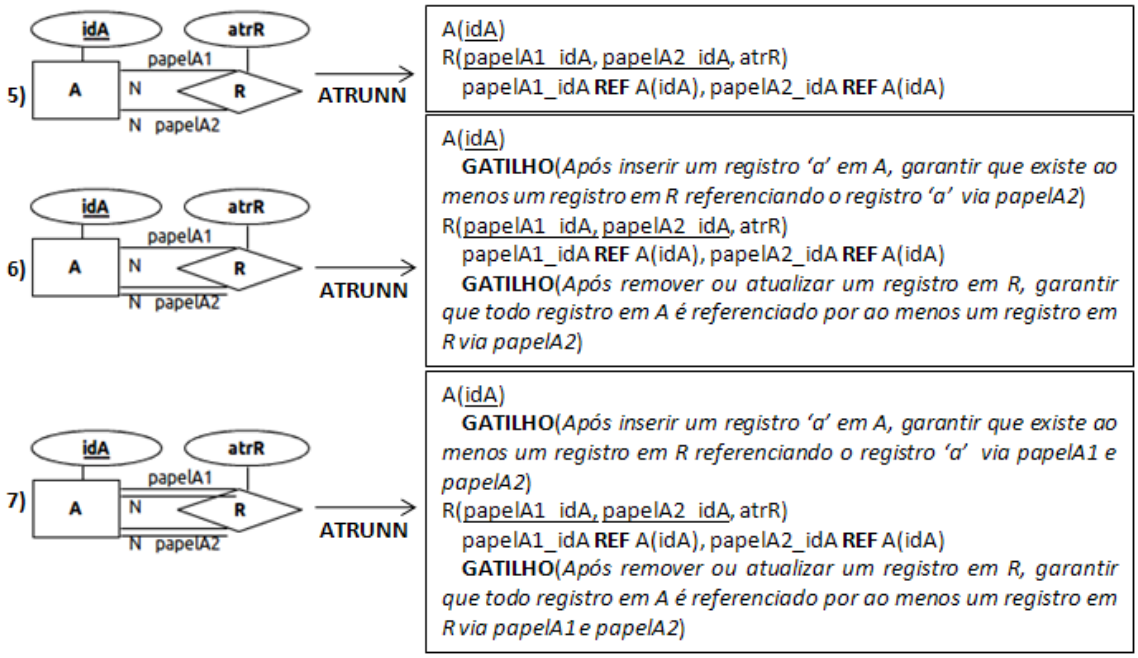
```
Input:      eerSchema : EERMM::Schema, rdbSchema: IMM::Schema;
Output:     rdbSchema: IMM::Schema;
Variables: tableA1, tableA2, tableNN : IMM::BaseTable;
           relationship : EERMM::Relationship;
           relationshipLinkA1, relationshipLinkA2 : EERMM::RelationshipLink;
           foreignKeyA1, foreignKeyA2 : IMM::ForeignKey;
           node : EERMM::Node;
           primaryKey : IMM::PrimaryKey;

1.   for each node ∈ eerSchema.nodes[] do
2.     if (node.isTypeOf(EERMM::RelationShip) = true) then
3.       relationship ← node;
4.       if(relationship.relationshipsLinks[].size() = 2 and
           relationship.associativeEntity = null and
           relationship.isIdentifier = false )  then
5.         relationshipLinkA1 ← relationship.relationshipsLinks[1];
6.         relationshipLinkA2 ← relationship.relationshipsLinks[2];
7.         if(isTable(relationshipLinkA1.source, rdbSchema) = true and
             isTable(relationshipLinkA2.source, rdbSchema) = true) then
8.           tableA1 ← getTable(relationshipLinkA1.source, rdbSchema);
9.           tableA2 ← getTable(relationshipLinkA2.source, rdbSchema);
```

```
10.          if(tableA1 = tableA2 and
               relationshipLinkA1.cardinality = "N" and
               relationshipLinkA2.cardinality = "N" ) then
11.            tableNN.name ← relationship.name;
12.            foreignKeyA1.referencedTable ←tableA1;
13.            foreignKeyA1.uniqueConstraint ← tableA1.primaryKey;
14.            foreignKeyA1.members[].add(
                 relationshipLinkA1.role + tableA1.primaryKey.columns[]);
15.            foreignKeyA2.referencedTable ←tableA2;
16.            foreignKeyA2.uniqueConstraint ← tableA2.primaryKey;
17.            foreignKeyA2.members[].add(
                 relationshipLinkA2.role + tableA2.primaryKey.columns[]);
18.            primaryKey.members[].add(relationshipLinkA1.role + tableA1.primaryKey.columns[]);
19.            primaryKey.members[].add(relationshipLinkA2.role + tableA2.primaryKey.columns[]);
20.            tableNN.constraints[].add(foreignKeyA1);
21.            tableNN.constraints[].add(foreignKeyA2);
22.            tableNN.constraints[].add(primaryKey);
23.            tableNN.columns[].add(relationshipLinkA1.role + tableA1.primaryKey.columns[]);
24.            tableNN.columns[].add(relationshipLinkA2.role + tableA1.primaryKey.columns[]);
25.            if( relationshipLinkA1.participation = "TOTAL" ) then
26.              tableA1.triggers.add("Após inserir um registro 'a' em tabela A, garantir que existe ao
                       menos um registro em tabela NN referenciando o registro 'a' via papelA1 ");
27.              tableNN.triggers.add("Após remover ou atualizar um registro em tabela NN, garantir
                                  que todo registro em tabela A é referenciado via papelA1 por
                                  ao menos um registro em tabela NN");
28.            end if
29.            if( relationshipLinkA2.participation = "TOTAL" ) then
30.              tableA2.triggers.add("Após inserir um registro 'a' em tabela A, garantir que existe ao
                       menos um registro em tabela NN referenciando o registro 'a' via o papelA2 ");
31.              tableNN.triggers.add("Após remover ou atualizar um registro em tabela NN, garantir
                                  que todo registro em tabela A é referenciado via o papelA2 por
                                  ao menos um registro em tabela NN");
32.            end if
33.            tableNN ← ATA1(relationship, tableNN);
34.            tableNN ← ATA2(relationship, tableNN);
35.            rdbSchema.add(tableNN);
36.            rdbSchema.add( ATA3(relationship, tableNN) );
37.            rdbSchema.update(tableA1);
38.          end if
39.        end if
40.      end if
41.    end if
42. end for
43. return rdbSchema;
```

O Algoritmo 10 é semelhante ao Algoritmo 7, as diferenças iniciam com o fato de que o Algoritmo 10 testa se o Relacionamento é Unário (cf. linha 10) e as Colunas das Chaves Estrangeiras e Chaves Primárias são criadas com um prefixo corresponde ao Papel das ligações do Relacionamento (cf. linhas 14, 17, 18 e 19). Na Figura 19, o Algoritmo ATRUNN é aplicado sobre os três possíveis casos de Cardinalidade e Participação em um Relacionamento

Unário N:N. A seguir, apresenta-se a regra para transformação de Relacionamentos N-ário.

Figura 19 - Transformação de Relacionamento Unário N:N.



Fonte: Autor (2015).

4.2.5 Regras para transformação de Relacionamento N-ário

A Regra para Transformação de Relacionamento N-ário (RTRN) é especificada usando apenas o Algoritmo ATRN (cf. Algoritmo 11), o qual faz uso dos algoritmos ATA1, ATA2 e ATA3. Vale destacar que, assim como a RTRB e RTRU, a especificação destes algoritmos requer a definição de Gatilhos para garantir a Participação Total das Entidades envolvidas no Relacionamento.

Algoritmo 11: ATRN – Transformação Relacionamento N-ário

```
Input:      eerSchema : EERMM::Schema, rdbSchema: IMM::Schema;
Output:     rdbSchema: IMM::Schema;
Variables:  table, tableNN : IMM::BaseTable;
            relationship : EERMM::Relationship;
            relationshipLink, relationshipLinkSelected : EERMM::RelationshipLink;
            foreignKey: IMM::ForeignKey;
            node : EERMM::Node;
            primaryKey : IMM::PrimaryKey;
            uniqueConstraint : IMM::UniqueConstraint;
            countRelationshipLinkOne : Int;
            hasDependencies : Boolean;
```

```
1.  for each node ∈ eerSchema.nodes[] do
2.    if (node.isTypeOf(EERMM::RelationShip) = true) then
3.      relationship ← node;
4.      if(relationship.relationshipsLinks[].size() > 2 and
          relationship.associativeEntity = null and
          relationship.isIdentifier = false )  then
5.        hasDependencies = false;
6.        for each relationshipLink ∈ relationship.relationshipsLinks[] do
7.          if(isTable(relationshipLink.source, rdbSchema) = false) then
8.            hasDependencies = true;  /* A tabela não existe no esquema */
9.          end if
10.       end for
11.       if(hasDependencies = false) then
12.         countRelationshipLinkOne ← relationship. getCountOfOneLinks();
13.         relationshipLinkSelected ← relationship.relationshipsLinks[].getLink("ChosenLinkTrue");
14.         tableNN ← relationship.name;
15.         for each relationshipLink ∈ relationship.relationshipsLinks[] do
16.           table ← getTable(relationshipLink.source, rdbSchema);
17.           foreignKey.referencedTable ←table;
18.           foreignKey.uniqueConstraint ← table.primaryKey;
19.           foreignKey.members[].add(table.primaryKey.columns[]);
20.           foreignKey.setNullable(true);
21.           tableNN.constraints[].add(foreignKey);
22.           if (countRelationshipLinkOne > 1 ) then
23.             if( relationshipLink.cardinality = "1" ) then
24.               if( relationshipLink = relationshipLinkSelected ) then
25.                 uniqueConstraint.members[].add(table.primaryKey.columns[]);
26.               else
27.                 primaryKey.members[].add(table.primaryKey.columns[]);
28.               end if
29.             else
30.               primaryKey.members[].add(table.primaryKey.columns[]);
31.               uniqueConstraint.members[].add(table.primaryKey.columns[]);
32.             end if
33.           else
34.             if (relationshipLink.cardinality = "N") then
35.               primaryKey.members[].add(table.primaryKey.columns[]);
36.             end if
37.           end if
38.           if( relationshipLink.participation = "TOTAL" ) then
39.             table.triggers.add("Após inserir um registro 'a' em tabela A, garantir que existe ao
                                    menos um registro em tabela NN referenciando o registro 'a' ");
40.             tableNN.triggers.add("Após remover ou atualizar um registro em tabela NN, garantir
                                      que todo registro em tabela A é referenciado por ao menos um
                                      registro em tabela NN");
41.             rdbSchema.update(table);
42.           end if
43.         end for
44.         tableNN.constraints[].add(primaryKey);
45.         tableNN.constraints[].add(uniqueConstraint);
46.         tableNN.columns[].add(table.primaryKey.columns[]);
47.         tableNN ← ATA1(relationship, tableNN);
48.         tableNN ← ATA2(relationship, tableNN);
49.         rdbSchema.add( ATA3(relationship, tableNN) );
50.         rdbSchema.add(tableNN);
51.       end if
```

```
52.       end if
53.    end if
54. end for
55. return rdbSchema;
```

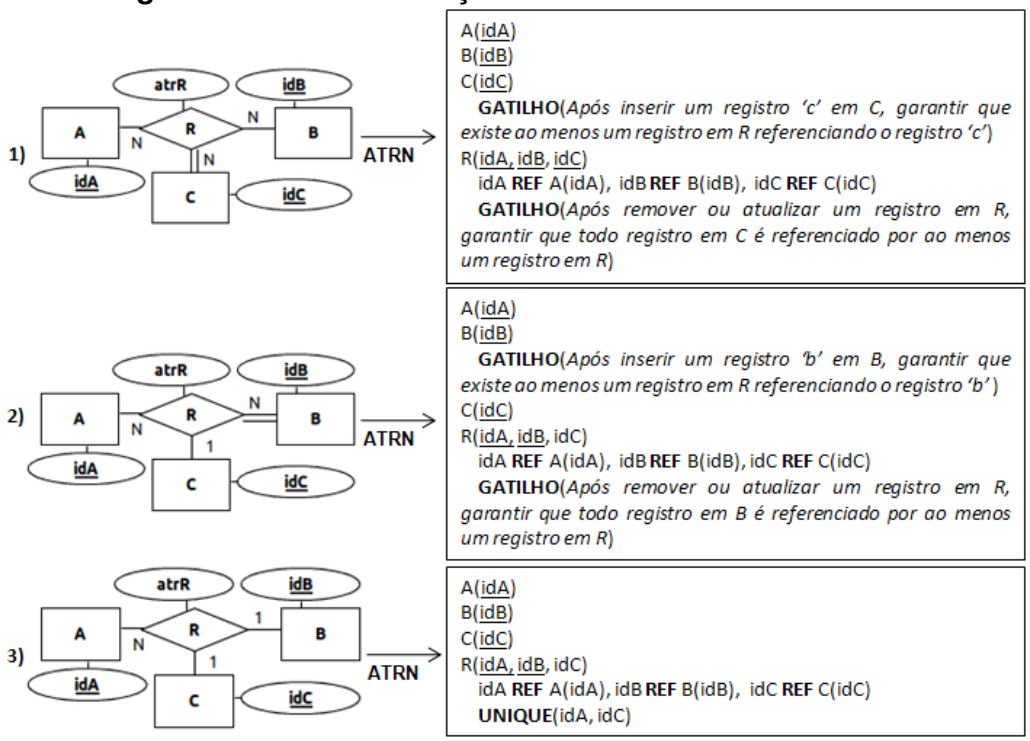
O inicio do Algoritmo 11 é semelhante ao apresentado no algoritmo 10 (cf. linhas 1 a 3), diferindo no fato que no Algoritmo 11 verifica se o Relacionamento é N-ário (cf. linha 4). Em seguida, a variável *hasDependencies* é inicializada com falso e é verificado se os Elementos ligados ao Relacionamento já foram transformados em Tabelas no esquema Relacional (cf. linha 7). Caso o resultado seja falso para algum Elemento, atribui-se o valor verdadeiro a *hasDependencies* (cf. linha 8). Em seguida, é verificado se o valor de *hasDependencies* é falso (cf. linha 11). Caso o resultado da verificação seja falso, o Relacionamento N-ário não é transformado. Do contrario, a transformação segue com a atribuição do valor da quantidade de Ligações de Relacionamento com Cardinalidade 1 em *countRelationshipLinkOne* (cf. linha 12), a atribuição da Ligação de Relacionamento escolhida para manter a Cardinalidade do Relacionamento para a variável *relationshipLinkSelected* (cf. linha 13) e é criada uma Tabela correspondente ao Relacionamento (cf. linha 14). Depois disso, para cada ligação do Relacionamento define-se: 1) a Tabela criada a partir do Elemento correspondente a Ligação de Relacionamento (i.e., convencionada como *table*) (cf. linha 16); 2) uma Chave Estrangeira que referencia esta Tabela (cf. linha 17); 3) a Chave Estrangeira aponta para a Chave Primária desta Tabela (cf. linha 18); 4) quais são os campos da Chave Primária que formam a Chave estrangeira (cf. linha 19); e 5) que a Chave Estrangeira é não anulável (cf. linha 20) e adicionada à coleção de restrições da Tabela criada a partir do Relacionamento (cf. linha 21).

Para definir a Chave Primária de *table*, primeiro é verificado se existe mais de uma ligação do Relacionamento com Cardinalidade 1 (cf. linha 22). Caso o resultado da verificação seja verdadeiro, é testado se a ligação do Relacionamento tem Cardinalidade 1 (cf. linha 23), se o resultado do teste também for verdadeiro e a ligação do Relacionamento for igual à *relationshipLinkSelected*, as Colunas da Chave Primária de *table* são adicionadas na restrição de unicidade, caso contrário, estas Colunas são adicionadas à Chave Primária (cf. linhas 24 a 28). Se a ligação do

Relacionamento não tiver Cardinalidade 1, as Colunas que compõe a Chave Primária de *table* são adicionadas na restrição de unicidade e na Chave Primária (cf. linhas 30 e 31). Caso não exista mais de uma ligação de Relacionamento com Cardinalidade 1, todas as Colunas das Chaves Primárias das Tabelas criadas a partir das Entidades correspondentes as ligações de Relacionamento com Cardinalidade N são adicionadas a Chave Primária da Tabela criada a partir do Relacionamento (cf. linhas 34 e 35).

Os passos seguintes do Algoritmo 10 tratam da Participação Total do Relacionamento. Neste caso, devem-se criar dois Gatilhos: um na Tabela referenciada pela Chave Estrangeira (cf. linha 39) e o outro na Tabela que contém a Chave Estrangeira (cf. linha 40). Por fim, a Chave Primária e a restrição de unicidade são adicionadas como restrições da Tabela correspondente ao Relacionamento (cf. linha 44 a 46), os possíveis atributos do Relacionamento são transformados (cf. linhas 47 a 49), e o esquema Relacional é retornado (cf. linha 55). Na Figura 20, o Algoritmo ATRN é aplicado sobre três possíveis casos de Cardinalidade e Participação em um Relacionamento N-ário. A seguir, apresenta-se a regra para transformação de Relacionamento Identificador.

Figura 20 - Transformação de Relacionamento N-ário.



Fonte: Autor (2015).

4.2.6 Regras para transformação de Relacionamento Identificador

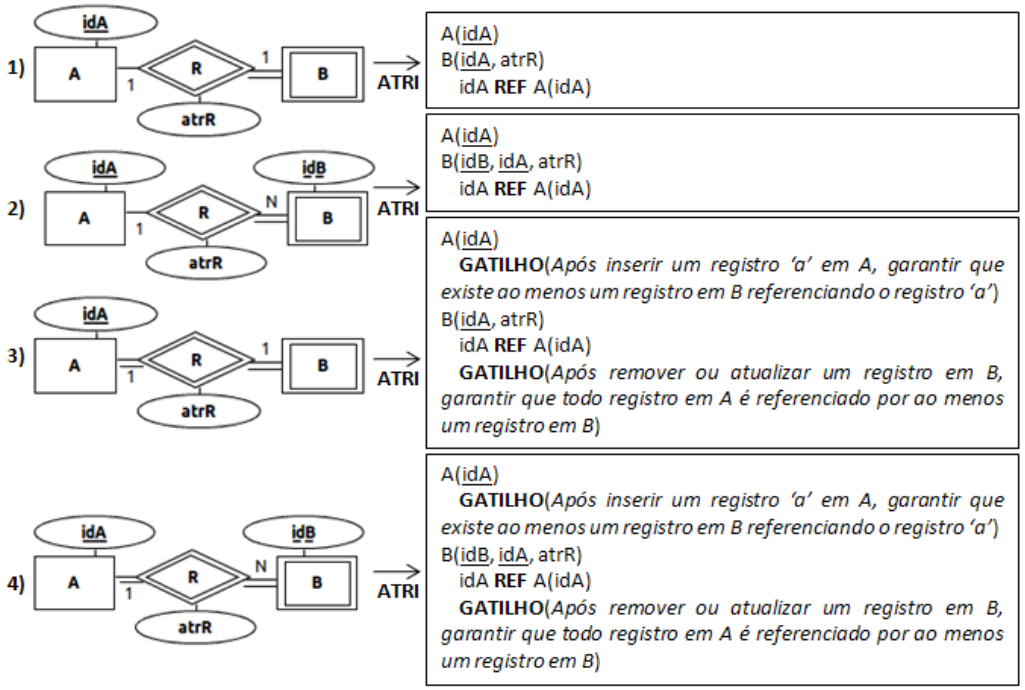
A Regra para Transformação de Relacionamento Identificador (RTRI) é especificada usando apenas o Algoritmo ATRI (cf. Algoritmo 12), o qual faz uso dos algoritmos ATA1, ATA2 e ATA3.

Algoritmo 12: ATRI - Transformação de Relacionamento Identificador

```
Input:      eerSchema : EERMM::Schema, rdbSchema: IMM::Schema;
Output:     rdbSchema: IMM::Schema;
Variables: tableA, tableB : IMM::BaseTable;
           relationship : EERMM::Relationship;
           relationshipLinkA, relationshipLinkB : EERMM::RelationshipLink;
           primaryKey : IMM::PrimaryKey;
           foreignKey : IMM::ForeignKey;
           node : EERMM::Node;

1.  for each node ∈ eerSchema.nodes[] do
2.    if (node.isTypeOf(EERMM::RelationShip) = true) then
3.      relationship ← node;
4.      if( relationship.isIdentifier = true ) then
5.        relationshipLinkA ← relationship.relationshipsLinks[].getLink("isIdentifierFalse");
6.        relationshipLinkB ← relationship.relationshipsLinks[].getLink("isIdentifierTrue");
7.        if( isTable(relationshipLinkA.source, rdbSchema) = true) then
8.          tableA ← getTable(relationshipLinkA.source, rdbSchema);
9.          tableB.name ← relationshipLinkB.source.name;
10.         tableB ← ATA1(relationshipLinkB.source, tableB);
11.         tableB ← ATA2(relationshipLinkB.source, tableB);
12.         foreignKey.referencedTable ← tableA;
13.         foreignKey.uniqueConstraint ← tableA.primaryKey;
14.         foreignKey.members[].add( tableA.primaryKey.columns[]);
15.         tableB.columns[].add(tableA.primaryKey.columns[]);
16.         tableB.constraints[].add(foreignKey);
17.         primaryKey.members[].add(tableA.primaryKey.columns[]);
18.         tableB.constraints[].add(primaryKey);
19.         if(relationshipLinkA.participation = "TOTAL") then
20.           tableA.triggers.add(" Após inserir um registro 'a' em tabela A, garantir que existe ao
                                  menos um registro em tabela B referenciando o registro 'a' ");
21.           tableB.triggers.add(" Após remover ou atualizar um registro em tabela B, garantir
                                  que todo registro em tabela A é referenciado por ao menos um
                                  registro em tabela B");
22.         end if
23.         tableB ← ATA1(relationship, tableB);
24.         rdbSchema.add( ATA3(relationship, tableB) );
25.         rdbSchema.add(tableB);
26.         rdbSchema.update(tableA);
27.       end if
28.     end if
29.   end if
30. end for
31. return rdbSchema;
```

O Algoritmo 12 é semelhante ao Algoritmo 6, com a diferença que no Algoritmo 12 convenciona-se que as Entidades correspondentes às ligações marcadas como "*isIdentifierFalse"* e "*isIdentifierTrue*" são manipuladas, respectivamente, pelos objetos *relationshipLinkA* e *relationshipLinkB* (cf. linhas de 5 e 6). Além disso, é criada uma Tabela para o Elemento correspondente a *relationshipLinkB* (cf. linhas 9 a 11) e as Colunas criadas a partir da Chave Estrangeira também são membros da Chave Primária da Tabela criada a partir do Elemento correspondente a *relationshipLinkB* (cf. linhas 17 e 18). Na Figura 21, o Algoritmo ATRI é aplicado sobre os quatro possíveis casos de Cardinalidade e Participação em um Relacionamento Identificador. A seguir, apresenta-se a regra para transformar Heranças.

Figura 21 - Transformação de Relacionamento Identificador.



Fonte: Autor (2015).

4.2.7 Regras para transformação de Herança

A Regra para Transformação de Herança (RTH) é modularizada em dois algoritmos: ATHD (cf. Algoritmo 13) e ATH (cf. Algoritmo 14), os quais transformam Heranças Diretas e Disjuntas, respectivamente. Vale destacar que a especificação do Algoritmo ATH requer a definição de Gatilhos para garantir a Completude Total e Disjunção das Entidades envolvidas na Herança.

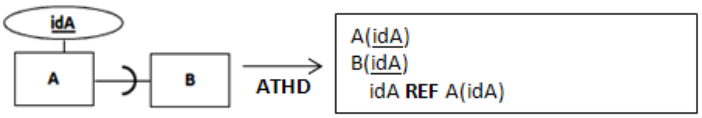
Algoritmo 13: ATHD – Transformação de Herança Direta

```
Input:      eerSchema : EERMM::Schema, rdbSchema: IMM::Schema;
Output:     rdbSchema: IMM::Schema;
Variables:  tableSuper, tableSub : IMM::BaseTable;
            primaryKey : IMM::PrimaryKey;
            foreignKey : IMM::ForeignKey;
            node : EERMM::Node;
            directInheritance:EERMM::DirectInheritanceLink;

1.  for each node ∈ eerSchema.nodes[] do
2.    if (node.isTypeOf(EERMM::DirectInheritanceLink) = true) then
3.      directInheritance ← node;
4.      if( isTable(directInheritance.source, rdbSchema) = true) then
5.        tableSuper ← getTable(directInheritance.source, rdbSchema);
6.        tableSub ← directInheritance.target.name;
7.        tableSub ← ATA1(directInheritance.target, tableSub);
8.        foreignKey.referencedTable ← tableSuper;
9.        foreignKey.uniqueConstraint ← tableSuper.primaryKey;
10.       foreignKey.members[].add(tableSuper.primaryKey.columns[]);
11.       primaryKey.members[].add(tableSuper.primaryKey.columns[]);
12.       tableSub.columns[].add(tableSuper.primaryKey.columns[]);
13.       tableSub.constraints[].add(foreignKey);
14.       tableSub.constraints[].add(primaryKey);
15.       rdbSchema.add(tableSub);
16.       rdbSchema.add( ATA3(directInheritance.target, tableSub) );
17.     end if
18.   end if
19. end for
20. return rdbSchema;
```

O Algoritmo 13 recebe como entrada um esquema EER a ser transformado e um esquema Relacional correspondente ao que já foi transformado. Por sua vez, a saída deste algoritmo corresponde ao esquema Relacional com as Heranças Diretas do esquema EER que possuem suas dependências estruturais satisfeitas transformados. A execução do algoritmo se inicia percorrendo cada Nó do Esquema EER modelado, testando e ajustando-o para que este corresponda a uma Herança Direta (cf. linhas de 1 a 3). Na sequência, para cada Herança Direta, é verificado se a Super-Entidade já foi transformada em Tabela no esquema Relacional (i.e., se a dependência estrutural foi satisfeita) (cf. linha 4). Se o resultado for verdadeiro: 1) cria-se uma Tabela com o mesmo nome da Sub-Entidade (cf. linha 6); 2) executa-se a regra RTA1 (cf. linhas 7); 3) é criada uma Chave Estrangeira que referencia a Tabela correspondente a Super-Entidade (cf. linha 8) que aponta para a Chave Primária desta Tabela (cf. linha 9); e 4) os campos da Chave Primária compõe a Chave Estrangeira (cf. linha 10). Em seguida, os campos definidos para a Chave Estrangeira são definidos como membros da Chave Primária da Tabela

(cf. linha 11). Por fim, a Chave Estrangeira e a Chave Primária são adicionadas à coleção de restrições da Tabela correspondente a Sub-Entidade (cf. linhas 13 e 14) e o esquema Relacional é retornado (cf. linha 20). Na Figura 22, é mostrado um exemplo de execução do Algoritmo ATHD. A seguir, apresenta-se o algoritmo para transformar Herança com Disjunção e Completude.

Figura 22 - Transformação de Herança Direta.



Fonte: Autor (2015).

Algoritmo 14: ATH – Transformação de Herança

```
Input:      eerSchema : EERMM::Schema, rdbSchema: IMM::Schema;
Output:     rdbSchema: IMM::Schema;
Variables:  tableSuper, tableSub : IMM::BaseTable;
            primaryKey : IMM::PrimaryKey;
            foreignKey : IMM::ForeignKey;
            node : EERMM::Node;
            inheritance : EERMM::Inheritance;
            inheritanceGL : EERMM::InheritanceGL;
            inheritanceSL : EERMM::InheritanceSL;
1.  for each node ∈ eerSchema.nodes[] do
2.    if (node.isTypeOf(EERMM::Inheritance) = true) then
3.      inheritance ← node;
4.      inheritanceGL ← inheritance.inheritanceGL;
5.      if( isTable(inheritanceGL.source, rdbSchema) = true) then
6.        tableSuper ←  getTable(inheritanceGL.source, rdbSchema);
7.        for each inheritanceSL ∈ inheritance.inheritanceSL[] do
8.          tableSub ← inheritanceSL.target.name;
9.          tableSub ← ATA1(inheritanceSL.target, tableSub);
10.         foreignKey.referencedTable ← tableSuper;
11.         foreignKey.uniqueConstraint ← tableSuper.primaryKey;
12.         foreignKey.members[].add( tableSuper.primaryKey.columns[]);
13.         primaryKey.members[].add( tableSuper.primaryKey.columns[]);
14.         tableSub.columns[].add(tableSuper.primaryKey.columns[]);
15.         tableSub.constraints[].add(foreignKey);
16.         tableSub.constraints[].add(primaryKey);
17.         if (inheritance.disjointness = "DISJOINT" and
              inheritance.completeness = "PARTIAL" ) then
18.           tableSub.triggers.add("Após inserir ou atualizar um registro na tabela Sub, garantir que
                                     todo registro em tabela Super é referenciado por somente um
                                     registro nas tabelas criadas a partir das sub-entidades ");
19.         end if
20.         if (inheritance.disjointness = "DISJOINT" and
              inheritance.completeness = "TOTAL" ) then
21.           tableSuper.triggers.add("Após inserir um registro 'g' em tabela Super, garantir que
                                       existe um registro nas tabelas criadas a partir das
                                       sub-entidades referenciando o registro 'g'");
22.           tableSub.triggers.add("Após inserir, excluir ou atualizar um registro na tabela Sub,
```

```
                                        garantir que todo registro na tabela Super é referenciado por
                                    somente um registro nas tabelas criadas a partir das sub-entidades");
23.          end if
24.          if (inheritance.disjointness = "OVERLAP" and
               inheritance.completeness = "TOTAL" ) then
25.            tableSuper.triggers.add("Após inserir um registro 'g' em tabela Super, garantir que
                                        existe ao menos um registro nas tabelas criadas a partir das
                                        sub-entidades referenciando o registro 'g'");
26.            tableSub.triggers.add("Após excluir ou atualizar um registro em tabela Sub, garantir
                                      que todo registro em tabela Super é referenciado por ao menos
                                      um registro nas tabelas criadas a partir das sub-entidades");
27.          end if
28.          rdbSchema.add(tableSub);
29.          rdbSchema.update(tableSuper);
30.        end for
31.      end if
32.    enf if
33. end for
34. return rdbSchema;
```

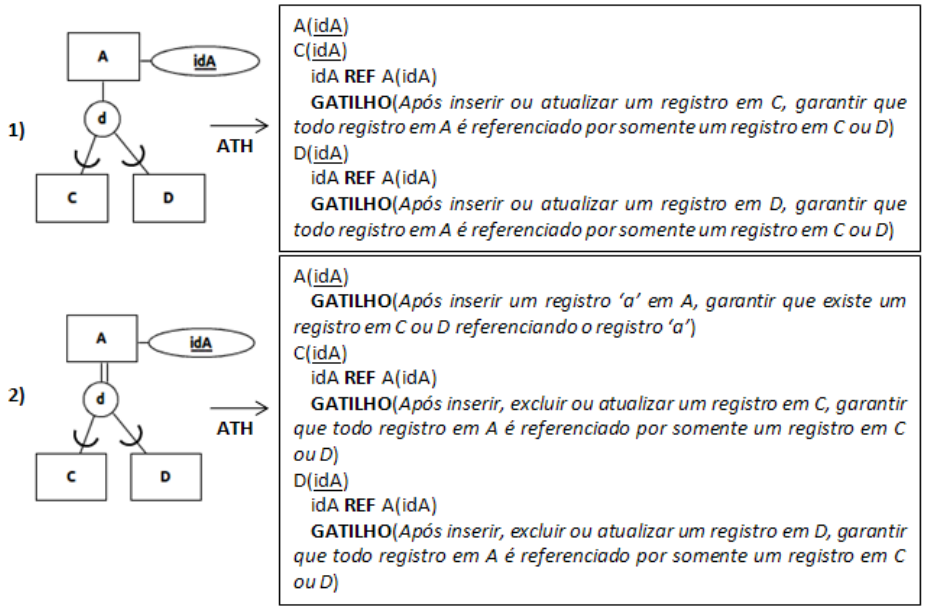
O Algoritmo 14 recebe como entrada um esquema EER a ser transformado e um esquema Relacional correspondente ao que já foi transformado. Por sua vez, a saída deste algoritmo corresponde ao esquema Relacional com as Heranças Disjuntas ou Sobrepostas do esquema EER que possuem suas dependências estruturais satisfeitas transformados. A execução do algoritmo se inicia percorrendo cada Nó do Esquema EER modelado, testando e ajustando-o para que este corresponda a uma Herança (cf. linhas de 1 a 3). Na sequência, é verificado se a Super-Entidade já existe como Tabela no esquema Relacional (i.e., dependências estruturais satisfeitas) (cf. linha 5). Caso o resultado for verdadeiro, para cada Sub-Entidade da Herança (cf. linha 7), define-se: 1) uma Tabela com o mesmo nome da Sub-Entidade (cf. linha 8); 2) executa-se a regra RTA1 (cf. linha 9); 3) é criada uma Chave Estrangeira que referencia a Tabela correspondente a Super-Entidade (cf. linha 10) que aponta para a Chave Primária desta Tabela (cf. linha 11); e 4) os campos da Chave Primária que formam a Chave Estrangeira (cf. linha 12). Em seguida, os campos definidos para a Chave Estrangeira são adicionados como membros da Chave Primária da Tabela (cf. linha 13) e a Chave Estrangeira e a Chave Primária são adicionadas a coleção de restrições da Tabela correspondente a Sub-Entidade (cf. linhas 15 e 16).

Os passos seguintes do Algoritmo 14 tratam da Disjunção (Disjunta e Sobreposta) e Completude (Total ou Parcial) da Herança. Se a Herança for

Disjunta e Parcial (cf. linha 17), deve-se criar Gatilhos nas Tabelas correspondentes as Sub-Entidades para garantir a restrição de Disjunção (cf. linha 18). Se a Herança for Disjunta e Total (cf. linha 20), devem-se criar dois tipos de Gatilhos, um na Tabela correspondente a Super-Entidade (cf. linha 21) para garantir a Completude Total e um Gatilho para cada Tabela correspondente as Sub-Entidades (cf. linha 22) para garantir as características Total e Disjunta. Se a Herança for Sobreposta e Total (cf. linha 24), devem-se criar dois tipos de Gatilhos, um na Tabela correspondente a Super-Entidade (cf. linha 25) e outro nas Tabelas correspondentes às Sub-Entidades (cf. linha 26), ambos para garantir a Completude Total. Ademais, a Tabela correspondente à Super-Entidade é atualizada no esquema Relacional e a Tabela correspondente à Sub-Entidade é adicionada ao esquema Relacional (cf. linhas 28 e 29). Por fim, e o esquema Relacional é retornado (cf. linha 34).

Na Figura 23, o Algoritmo ATH é aplicado sobre os dois possíveis casos de Completude em Herança Disjunta.

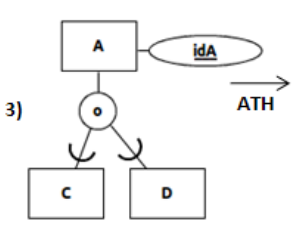
Figura 23 - Transformação de Herança Disjunta.



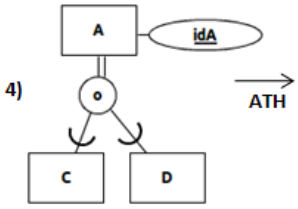
Fonte: Autor (2015).

Na Figura 24, o Algoritmo ATH é aplicado sobre os dois possíveis casos de Completude em Herança Sobreposta. A seguir, apresenta-se a regra para transformação de Categoria.

Figura 24 - Transformação de Herança Sobreposta.



A(idA)
C(idA)
idA **REF** A(idA)
D(idA)
idA **REF** A(idA)



A(idA)
GATILHO(*Após inserir um registro 'a' em A, garantir que existe ao menos um registro em C ou D referenciando o registro 'a'*)
C(idA)
idA **REF** A(idA)
GATILHO(*Após excluir ou atualizar um registro em C, garantir que todo registro em A é referenciado por ao menos um registro em C ou D*)
D(idA)
idA **REF** A(idA)
GATILHO(*Após excluir ou atualizar um registro em D, garantir que todo registro em A é referenciado por ao menos um registro em C ou D*)

Fonte: Autor (2015).

4.2.8 Regras para transformação de Categoria

A Regra para Transformação de Categoria (RTC) é especificada usando apenas o Algoritmo ATC (cf. Algoritmo 15), o qual faz uso dos algoritmos ATA1, ATA2 e ATA3.

Algoritmo 15: ATC – Transformação de Categoria

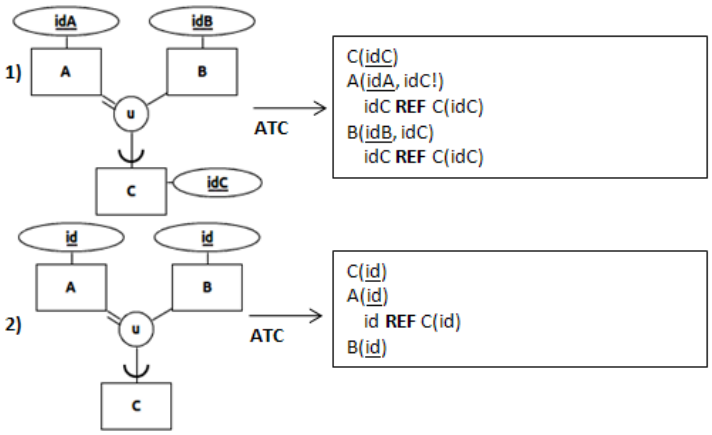
```
Input:      eerSchema : EERMM::Schema, rdbSchema: IMM::Schema;
Output:     rdbSchema: IMM::Schema;
Variables: tableSuper, tableSub : IMM::BaseTable;
           primaryKey : IMM::PrimaryKey;
           foreignKey : IMM::ForeignKey;
           node : EERMM::Node;
           category : EERMM::Category;
           categoryGL: EERMM::CategoryGL;
           categorySL : EERMM::CategorySL;
           hasDependencies: Boolean;

1.  for each node ∈ eerSchema.nodes[] do
2.    if (node.isTypeOf(EERMM::Category) = true) then
3.      category ← node;
4.      hasDependencies = false;
5.      for each categoryGL ∈ category.categoryGL [] do
6.        if(isTable(categoryGL.source, rdbSchema) = false) then
7.          hasDependencies = true;        /* A tabela não existe no esquema */
8.        end if;
9.      end for;
10.     if(hasDependencies = false) then
11.       categorySL ← category.categorySL;
12.       for each categoryGL ∈ category.categoryGL[] do
13.         tableSuper ←  getTable(categoryGL.source, rdbSchema);
14.         if( isTable(categorySL.source, rdbSchema) = false ) then
15.           tableSub ← categorySL.source.name;
16.           tableSub ← ATA1(categorySL.source, tableSub);
17.           tableSub ← ATA2(categorySL.source, tableSub);
18.           if ( tableSub.constraints[].size() = 0 ) then
```

```
19.                 primaryKey.members[].add( tableSuper.primaryKey.columns[]);
20.                 tableSub.constraints[].add(primaryKey);
21.                 tableSub.columns[].add(tableSuper.primaryKey.columns[]);
22.               end if;
23.               rdbSchema.add(tableSub);
24.             else
25.                tableSub ← getTable(categorySL.source, rdbSchema);
26.             end if;
27.             foreignKey.referencedTable ← tableSub;
28.             foreignKey.uniqueConstraint ← tableSub.primaryKey;
29.             foreignKey.members[].add( tableSub.primaryKey.columns[]);
30.             if ( categoryGL.completeness = "TOTAL" ) then
31.               foreignKey.setNullable(false);
32.             else
33.                foreignKey.setNullable(true);
34.             end if;
35.             if(tableSub. primaryKey [] <>  tableSuper. primaryKey []) then
36.               tableSuper.columns[].add(tableSub.primaryKey.columns[]);
37.             end if;
38.             tableSuper.constraints[].add(foreignKey);
39.             rdbSchema.update(tableSuper);
40.          end for
41.       end if
42.    enf if
43. end for
44. return rdbSchema;
```

O Algoritmo 15 recebe como entrada um esquema EER a ser transformado e um esquema Relacional correspondente ao que já foi transformado. Por sua vez, a saída deste algoritmo corresponde ao esquema Relacional com as Categorias do esquema EER que possuem suas dependências estruturais satisfeitas transformados. A execução do algoritmo se inicia percorrendo cada Nó do Esquema EER modelado, testando e ajustando-o para que este corresponda a uma Categoria (cf. linhas de 1 a 3). Em seguida, a variável *hasDependencies* é inicializada com falso e é verificado se as Super-Entidades da Categoria já foram transformadas em Tabelas no esquema Relacional (cf. linha 6). Caso o resultado seja falso para alguma Super-Entidade, é atribuído o valor verdadeiro para *hasDependencies* (cf. linha 7). Em seguida, é verificado se o valor de *hasDependencies* é falso (cf. linha 10). Sendo falso, a Categoria não é transformada. Do contrario, a transformação segue, e, para cada Super-Entidade da Categoria, define-se: 1) que *TableSuper* é a Tabela correspondente a Super-Entidade; e 2) Se a Tabela correspondente a Sub-Entidade ainda não foi criada, cria-se uma Tabela correspondente a Sub-Entidade (cf. linhas 14 a 23). Caso não seja definida Chave Primária na criação desta Tabela (cf. linha 18), a Chave Primária deve

ser igual à Chave Primária de *TableSuper*. Do contrário, é atribuída a *TableSub* a Tabela ao qual a Sub-Entidade foi transformada. Em seguida, defini-se: 1) uma Chave Estrangeira que referencia *TableSub* (cf. linha 27) que aponta para a Chave Primária desta Tabela (cf. linha 28); 2) quais são os campos da Chave Primária que formam a Chave estrangeira (cf. linha 29). Em seguida, verifica-se a Completude da Super-Entidade, caso seja Total a Chave Estrangeira não é anulável (cf. linha 31), do contrario é anulável (cf. linha 33). Se os campos da Chave Primária de *tableSub* forem iguais aos campos da Chave Primária de *tableSuper*, as Colunas da Chave Estrangeira são adicionadas a coleção de colunas de *tableSuper* e à coleção de restrições desta Tabela (cf. linha 38). Por fim, o esquema Relacional é retornado. Na Figura 25, o Algoritmo ATC é aplicado sobre dois exemplos de Categoria. A seguir, apresenta-se a regra para transformação de Entidade Associativa.

Figura 25 - Transformação de Categoria.



Fonte: Autor (2015).

4.2.9 Regras para transformação de Entidade Associativa

A Regra para Transformação de Entidade Associativa (RTEA) é modularizada em dois algoritmos: ATEARB (cf. Algoritmo 16) e ATEARN (cf. Algoritmo 17), os quais transformam Entidades Associativas conectadas com Relacionamentos Binários ou N-ários, respectivamente.

Algoritmo 16: ATEARB - Transformação de Entidade Associativa conectada com Relacionamento Binário

Input: eerSchema : EERMM::Schema, rdbSchema: IMM::Schema;
Output: rdbSchema: IMM::Schema;
Variables: tableA, tableB, tableEA : IMM::BaseTable;
relationship : EERMM::Relationship;

```
            associativeEntity : EERMM:: AssociativeEntity;
            relationshipLinkA, relationshipLinkB : EERMM::RelationshipLink;
            foreignKeyA, foreignKeyB : IMM::ForeignKey;
            node : EERMM::Node;
            primaryKey : IMM::PrimaryKey;
            uniqueConstraint : EERMM::UniqueConstraint;

1.  for each node ∈ eerSchema.nodes[] do
2.    if (node.isTypeOf(EERMM::AssociativeEntity) = true) then
3.      associativeEntity ← node;
4.      relationship ← associativeEntity.contains;
5.      if(relationship.relationshipsLinks[].size() = 2 and
          relationship.isIdentifier = false )  then
6.        relationshipLinkA ← relationship.relationshipsLinks[1];
7.        relationshipLinkB ← relationship.relationshipsLinks[2];
8.        if(isTable(relationshipLinkA.source, rdbSchema) = true and
            isTable(relationshipLinkB.source, rdbSchema) = true) then
9.          tableA ← getTable(relationshipLinkA.source, rdbSchema);
10.         tableB ← getTable(relationshipLinkB.source, rdbSchema);
11.         tableEA.name ← associativeEntity.name;
12.         foreignKeyA.referencedTable ←tableA;
13.         foreignKeyA.uniqueConstraint ← tableA.primaryKey;
14.         foreignKeyA.members[].add( tableA.primaryKey.columns[]);
15.         foreignKeyA.setNullable( false);
16.         foreignKeyB.referencedTable ←tableB;
17.         foreignKeyB.uniqueConstraint ← tableB.primaryKey;
18.         foreignKeyB.members[].add(tableB.primaryKey.columns[]);
19.         foreignKeyB.setNullable( false);
20.         tableEA.constraints[].add(foreignKeyA);
21.         tableEA.constraints[].add(foreignKeyB);
22.         tableEA.columns[].add(tableA.primaryKey.columns[]);
23.         tableEA.columns[].add(tableB.primaryKey.columns[]);
24.         if(relationshipLinkA.cardinality = "1" and relationshipLinkB.cardinality = "1" ) then
25.           if (relationshipLinkA.participation = "PARTIAL" and
                relationshipLinkB.chosenLink = false)  then
26.             primaryKey.members[].add(tableA.primaryKey.columns[]);
27.             uniqueConstraint.members[].add(tableB.primaryKey.columns[]);
28.           else
29.             primaryKey.members[].add(tableB.primaryKey.columns[]);
30.             uniqueConstraint.members[].add(tableA.primaryKey.columns[]);
31.           end if
32.         end if
33.         if(relationshipLinkA.cardinality = "1" and relationshipLinkB.cardinality = "N" ) then
34.           primaryKey.members[].add(tableA.primaryKey.columns[]);
35.         end if
36.         if(relationshipLinkA.cardinality = "N" and relationshipLinkB.cardinality = "1" ) then
37.           primaryKey.members[].add(tableB.primaryKey.columns[]);
38.         end if
39.         if(relationshipLinkA.cardinality = "N" and relationshipLinkB.cardinality = "N" ) then
40.           primaryKey.members[].add(tableA.primaryKey.columns[]);
41.           primaryKey.members[].add(tableB.primaryKey.columns[]);
42.         end if
43.           if( relationshipLinkA.participation = "TOTAL" ) then
44.             tableA.triggers.add("Após inserir um registro 'a' em tabela A, garantir que existe ao
                                    menos um registro em tabela EA referenciando o registro 'a' ");
45.             tableEA.triggers.add("Após remover ou atualizar um registro em tabela EA, garantir
                                    que todo registro em tabela A é referenciado  por ao menos um
```

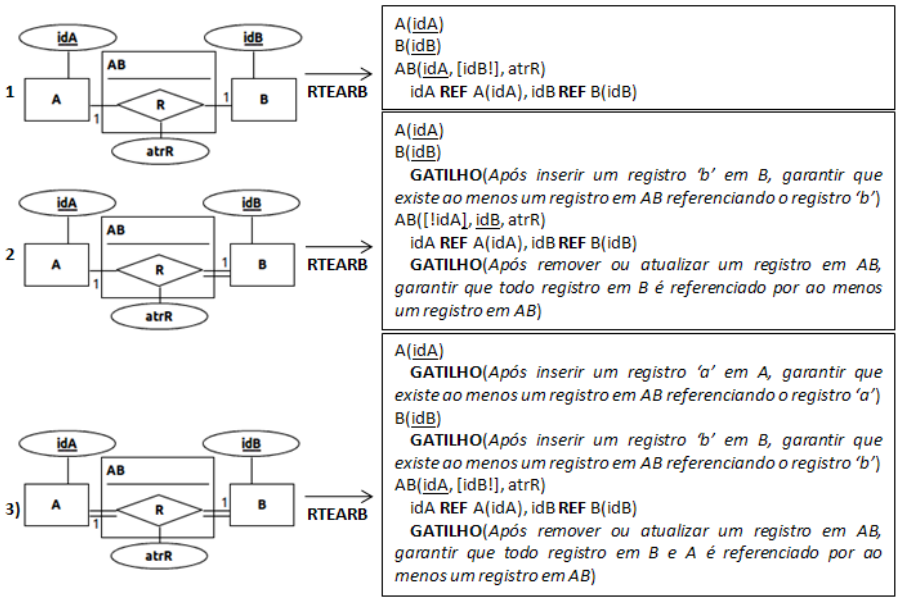
```
                                    registro em tabela EA”);
46.             end if
47.             if( relationshipLinkB.participation = “TOTAL” ) then
48.               tableB.triggers.add(“Após inserir um registro ‘a’ em tabela A, garantir que existe ao
                                    menos um registro em tabela EA referenciando o registro ‘a’”);
49.               tableEA.triggers.add(“Após remover ou atualizar um registro em tabela EA, garantir
                                    que todo registro em tabela A é referenciado  por ao menos um
                                    registro em tabela EA”);
50.             end if
51.             tableEA.constraints[].add(primaryKey);
52.             tableEA.constraints[].add(uniqueConstraint);
53.             tableEA ← ATA1(relationship, tableEA);
54.             tableEA ← ATA2(relationship, tableEA);
55.             rdbSchema.add(tableEA);
56.             rdbSchema.add( ATA3(relationship, tableEA) );
57.             rdbSchema.update(tableA);
58.             rdbSchema.update(tableB);
59.           end if
60.         end if
61.       end if
62.    end if
63. end for
64. return rdbSchema;
```

O Algoritmo 16 é semelhante ao Algoritmo 7, com a diferença que a Tabela criada corresponde à Entidade Associativa e as Chaves Primárias desta Tabela são decididas de acordo com a Cardinalidade do Relacionamento contido na Entidade Associativa. Se o Relacionamento for 1:1 (cf. linha 24), verifica-se se *relationshipLinkA* possui Participação Total e *relationshipLinkB* possui a propriedade *ChosenLink* igual a falso (cf. linha 25). Se o resultado da verificação for verdadeiro, é criado uma Chave Primária que tem como Colunas os campos da Tabela *tableA* e uma restrição de unicidade que tem como Colunas os campos da Tabela *tableB* (cf. linhas 26 e 27). Caso contrário, é criado uma Chave Primária que tem como Colunas os campos da Tabela *tableB* e uma restrição de unicidade que tem como Colunas os campos da Tabela *tableA* (cf. linhas 29 e 30). Em seguida, é testado se a Cardinalidade é 1:N (i.e, *relationshipLinkA* tem Cardinalidade 1 e *relationshipLinkB* tem Cardinalidade N), caso seja, a Chave Primária tem como Colunas os campos da Tabela *tableA* (cf. linha 34). Se a Cardinalidade for N:1 (i.e, *relationshipLinkA* tem Cardinalidade N e *relationshipLinkB* tem Cardinalidade 1), a Chave Primária tem como Colunas os campos da Tabela *tableB* (cf. linha 37). Além desses casos, é testado se a Cardinalidade é N:N, se for o caso, a Chave Primária é composta das Colunas das Chaves Primárias de *tableA* e

tableB (cf. linhas 40 e 41). Na sequência, a Chave Primária e as restrições de unicidades são adicionadas a coleção de restrições da Tabela *TableEA*. As colunas referentes aos Atributos do Relacionamento são adicionadas em *TableEA* (cf. linhas 51 a 54). Por fim, *TableEA* é adicionada ao esquema Relacional, as Tabelas *tableA* e *tableB* são atualizadas no esquema Relacional e este é retornado.

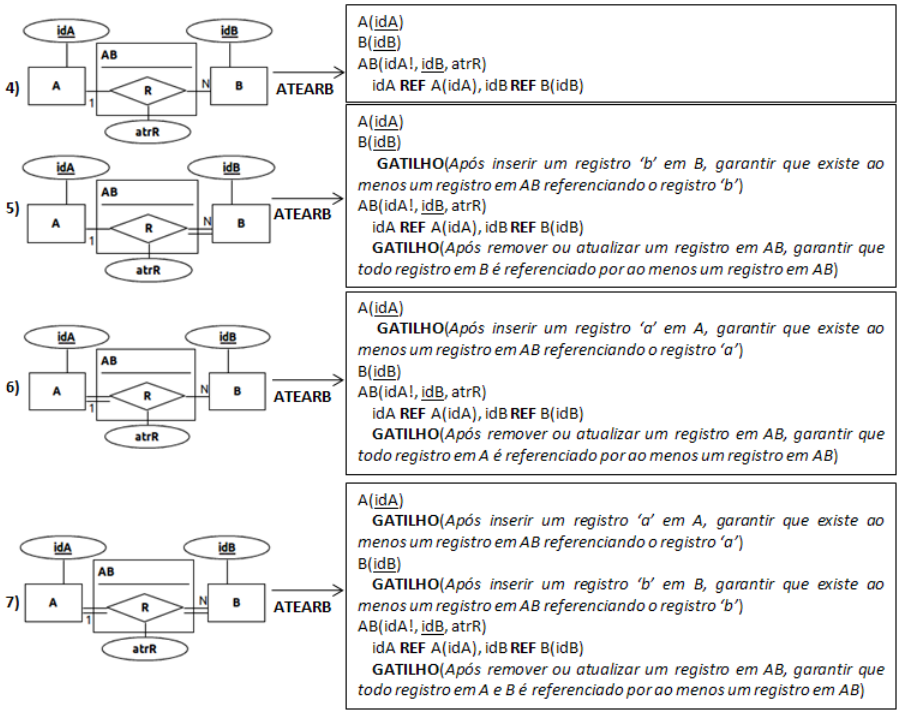
Na Figura 26, são mostrados exemplos da execução do Algoritmo ATEARB conectado com Relacionamentos Binário com Cardinalidade 1:1, na Figura 27, exemplos de Relacionamentos Binários com Cardinalidade 1:N e na Figura 28, exemplos com Relacionamentos Binários com Cardinalidade N:N. A seguir, apresenta-se o algoritmo para transformar Entidade Associativa com Relacionamento N-ário.

Figura 26 - Transformação de Entidade Associativa com Relacionamento Binário 1:1.



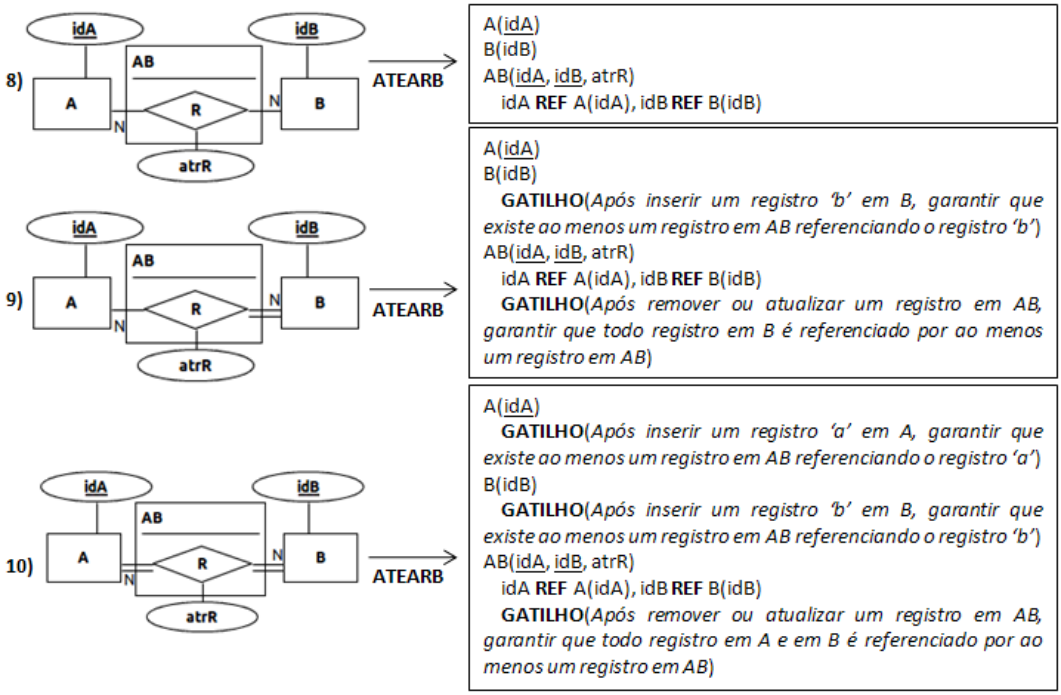
Fonte: Autor (2015).

Figura 27 - Transformação de Entidade Associativa com Relacionamento Binário 1:N.



Fonte: Autor (2015).

Figura 28 - Transformação de Entidade Associativa com Relacionamento Binário N:N.



Fonte: Autor (2015).

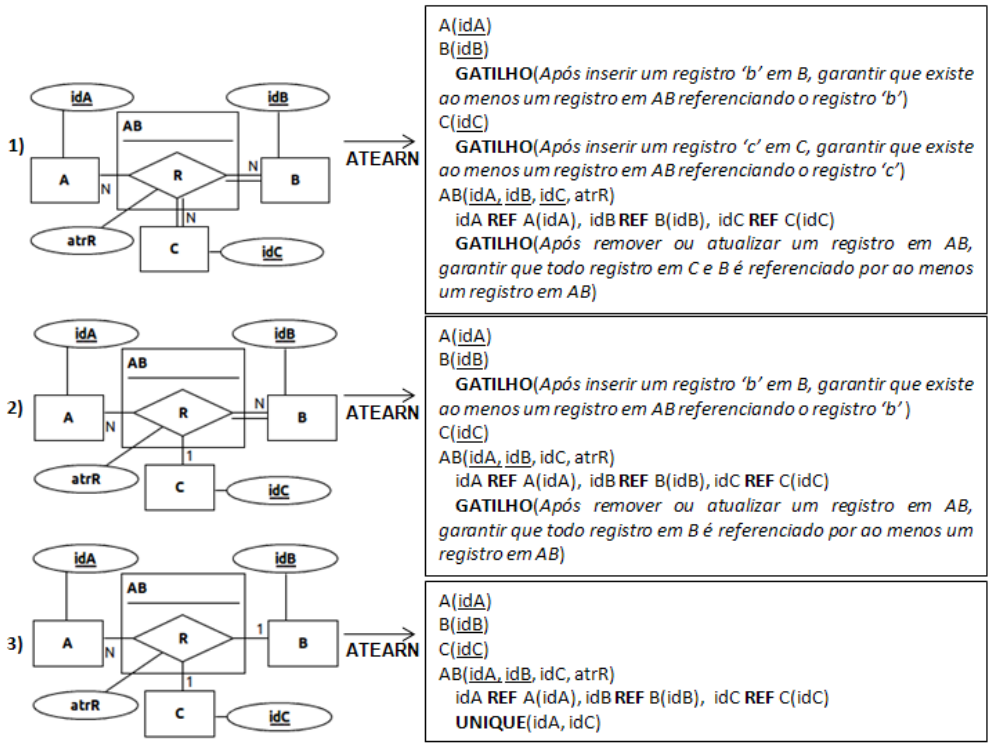
Algoritmo 17: ATEARN - Transformação de Entidade Associativa conectada com Relacionamento N-ário

```
Input:      eerSchema : EERMM::Schema, rdbSchema: IMM::Schema;
Output:     rdbSchema: IMM::Schema;
Variables: table, tableNN : IMM::BaseTable;
           relationship : EERMM::Relationship;
           associativeEntity : EERMM:: AssociativeEntity;
           relationshipLink, relationshipLinkSelected : EERMM::RelationshipLink;
           foreignKey: IMM::ForeignKey;
           node : EERMM::Node;
           primaryKey : IMM::PrimaryKey;
           uniqueConstraint : IMM::UniqueConstraint;
           countRelationshipLinkOne : Int;
           hasDependencies : Boolean;

1.  for each node ∈ eerSchema.nodes[] do
2.    if (node.isTypeOf(EERMM:: AssociativeEntity) = true) then
3.      associativeEntity ← node;
4.      relationship ← associativeEntity.contains;
5.      if(relationship.relationshipsLinks[].size() > 2 and
          relationship.isIdentifier = false )  then
6.        hasDependencies = false;
7.        for each relationshipLink ∈ relationship.relationshipsLinks[] do
8.          if(isTable(relationshipLink.source, rdbSchema) = false) then
9.            hasDependencies = true;  /* A tabela não existe no esquema */
10.         end if
11.       end for
12.       if(hasDependencies = false) then
13.         countRelationshipLinkOne ← relationship. getCountOfOneLinks();
14.         relationshipLinkSelected ← relationship.relationshipsLinks[].getLink("ChosenLinkTrue");
15.         tableNN ← associativeEntity.name;
16.         for each relationshipLink ∈ relationship.relationshipsLinks[] do
17.           table ← getTable(relationshipLink.source, rdbSchema);
18.           foreignKey.referencedTable ←table;
19.           foreignKey.uniqueConstraint ← table.primaryKey;
20.           foreignKey.members[].add(table.primaryKey.columns[]);
21.           foreignKey.setNullable(true);
22.           tableNN.constraints[].add(foreignKey);
23.           if (countRelationshipLinkOne > 1 ) then
24.             if( relationshipLink.cardinality = "1" ) then
25.               if( relationshipLink = relationshipLinkSelected ) then
26.                 uniqueConstraint.members[].add(table.primaryKey.columns[]);
27.               else
28.                 primaryKey.members[].add(table.primaryKey.columns[]);
29.               end if
30.             else
31.               primaryKey.members[].add(table.primaryKey.columns[]);
32.               uniqueConstraint.members[].add(table.primaryKey.columns[]);
33.             end if
34.           else
35.             if (relationshipLink.cardinality = "N") then
36.               primaryKey.members[].add(table.primaryKey.columns[]);
37.             end if
38.           end if
39.           if( relationshipLink.participation = "TOTAL" ) then
40.             table.triggers.add("Após inserir um registro 'a' em tabela A, garantir que existe ao
```

```
                                    menos um registro em tabela NN referenciando o registro 'a' ");
41.                 tableNN.triggers.add("Após remover ou atualizar um registro em tabela NN, garantir
                                    que todo registro em tabela A é referenciado por ao menos um
                                    registro em tabela NN");
42.                 rdbSchema.update(table);
43.               end if
44.             end for
45.             tableNN.constraints[].add(primaryKey);
46.             tableNN.constraints[].add(uniqueConstraint);
47.             tableNN.columns[].add(table.primaryKey.columns[]);
48.             tableNN ← ATA1(relationship, tableNN);
49.             tableNN ← ATA2(relationship, tableNN);
50.             rdbSchema.add( ATA3(relationship, tableNN) );
51.             rdbSchema.add(tableNN);
52.           end if
53.         end if
54.       end if
55. end for
56. return rdbSchema;
```

O Algoritmo 17 tem como base o Algoritmo 11, diferindo apenas no fato que a Tabela criada para o Relacionamento passa a ter o nome da Entidade Associativa. Na Figura 29, o Algoritmo ATEARN é aplicado sobre três exemplos de Entidade Associativa conectada com Relacionamento N-ário. A seguir, apresenta-se o Algoritmo para execução automática das regras de transformação do Catálogo.

Figura 29 - Transformação de Entidade Associativa com Relacionamento N-ário.

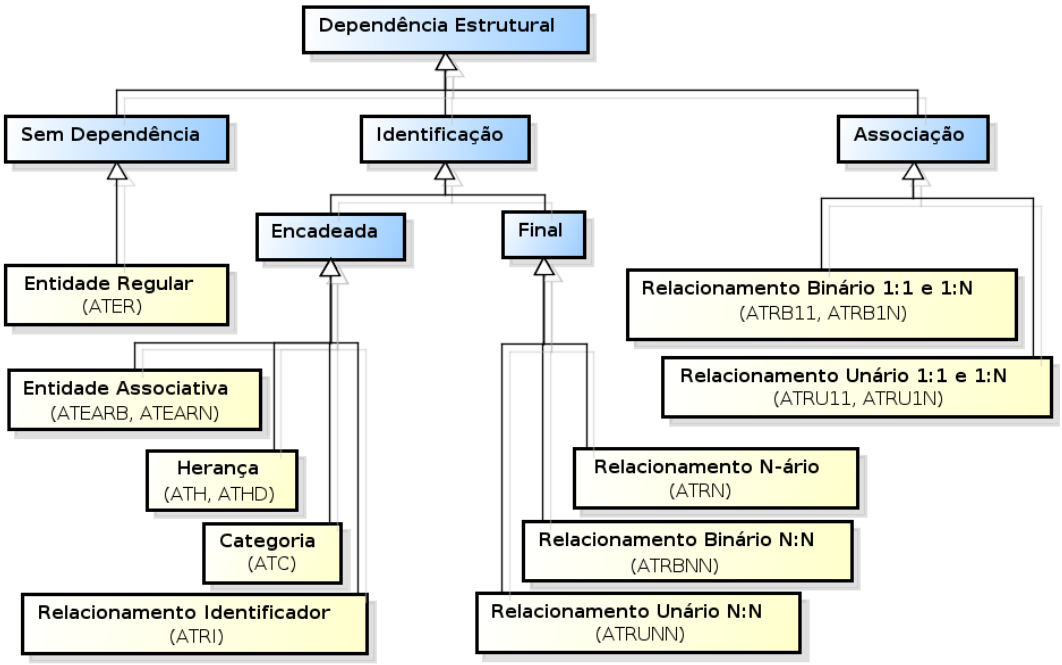


Fonte: Autor (2015).

4.3 Algoritmo para execução do Catálogo

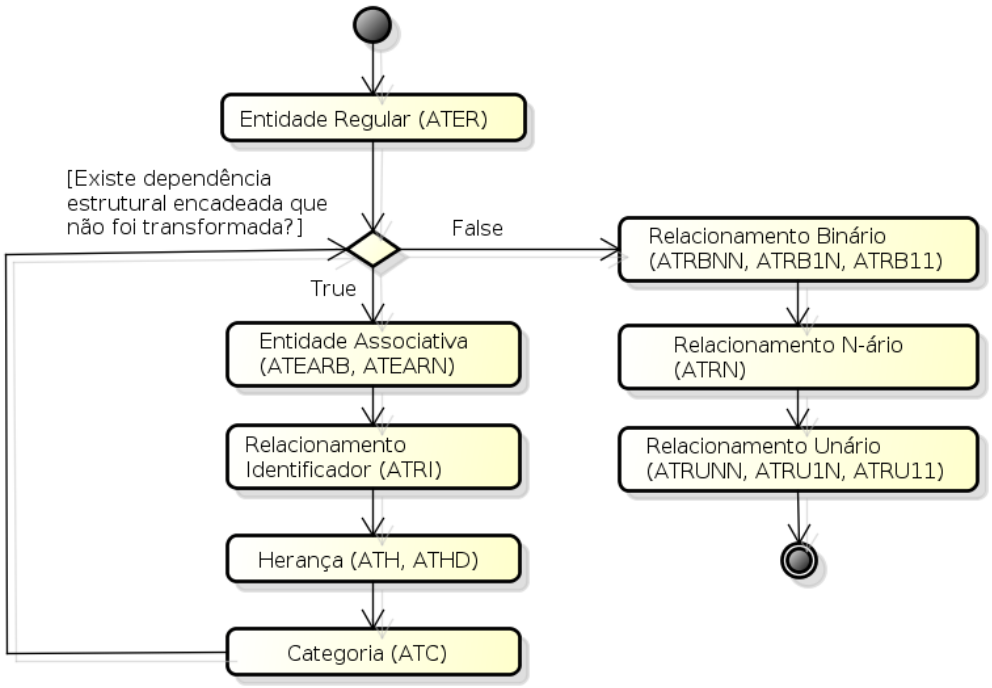
O Algoritmo para especificar a Transformação de Esquema EER em Esquema Relacional (ATEER2Rel) (cf. Algoritmo 18) define a lógica que permite executar as regras de transformações em uma ordem correta. Para isto, com base nas dependências estruturais discutidas na Seção 4.1, este Algoritmo inicia transformando as Entidades Regulares, pois estas não têm nenhuma dependência estrutural. Na sequência, o Algoritmo transforma as construções EER que podem ter dependência de identificação encadeada. Por fim, indistintamente, as construções com dependência de identificação final ou dependência de associação são executadas. A classificação dos algoritmos quanto à dependência estrutural é ilustrada no Diagrama de Classes na Figura 30 e ordem de execução destes algoritmos pelo Algoritmo ATEER2Rel é apresentada no Diagrama de Atividade na Figura 31. Vale ressaltar que os algoritmos que transformam construções encadeadas (i.e, Algoritmos ATH, ATHD, ATC, ATRI, ATEARB e ATEARN) podem ser executados várias vezes até que todas as construções encadeadas sejam transformadas. Isto é, estes algoritmos são executados enquanto existir construção com dependência de identificação encadeada sem ser transformada.

Figura 30 - Dependência Estrutural.



Fonte: Autor (2015).

Figura 31 – Algoritmo 18: Ordem de Execução.



Fonte: Autor (2015).

Para simplificar a especificação e explicação do Algoritmo 18 é utilizada a função *hasChainedDependencies(e:EERMM::Schema,r:IMM::Eschema):boolean* (cf. linha 2), que retorna um valor lógico e tem como entrada dois parâmetros: 1) o esquema EER a ser transformado e 2) a atual versão do esquema Relacional que está sendo transformado. Esta função é utilizada para verificar se existem construções EER com dependências estruturais de identificação encadeada que ainda não foram transformadas, retornando verdadeiro caso encontre alguma construção que ainda não foi transformada ou falso caso não existam mais estruturas a serem transformadas por estes algoritmos.

Algoritmo 18: ATEER2Rel –Transformacão de Esquema EER em Esquema Relacional

Input: eerSchema : EERMM::Schema;
Output: rdbSchema: IMM::Schema;

```
1. rdbSchema ← ATER(eerSchema);                          /* Entidade Regular*/
2. while (hasChainedDependencies (eerSchema, rdbSchema)  = true) do
3.    rdbSchema ← ATRI(eerSchema, rdbSchema);        /* Relacionamento Identificador*/
4.    rdbSchema ← ATH(eerSchema, rdbSchema);         /* Herança Disjunta ou Sobreposta*/
5.    rdbSchema ← ATHD(eerSchema, rdbSchema);        /* Herança Direta*/
6.    rdbSchema ← ATC(eerSchema, rdbSchema);          /* Categoria*/
7.    rdbSchema ← ATEARB(eerSchema, rdbSchema);      /* Entidade associativa Binário*/
8.    rdbSchema ← ATEARN(eerSchema, rdbSchema);     / * Entidade associativa N-ário*/
9. end while
```

```
10. rdbSchema ← ATRB11(eerSchema, rdbSchema);      /* relacionamento Binário 1:1*/
11. rdbSchema ← ATRB1N(eerSchema, rdbSchema);      /* Relacionamento Binário 1:N*/
12. rdbSchema ← ATRBNN(eerSchema, rdbSchema);      /* Relacionamento Binário N:N*/
13. rdbSchema ← ATRN(eerSchema, rdbSchema);        /* Relacionamento N-ário */
14. rdbSchema ← ATRU11(eerSchema, rdbSchema);      /* Relacionamento Unário 1:1*/
15. rdbSchema ← ATRU1N(eerSchema, rdbSchema);      /* Relacionamento Unário 1:N*/
16. rdbSchema ← ATRUNN(eerSchema, rdbSchema);      /*  Relacionamento Unário N:N*/
17. return rdbSchema;
```

No Algoritmo 18, tem-se como entrada um esquema EER a ser transformado e como saída o esquema Relacional correspondente ao esquema EER transformado. O corpo do algoritmo inicia com a execução do algoritmo para transformar Entidades Regulares (ATER - cf. linha 1). Na sequência, em uma estrutura de laço, a função *hasChainedDependencies* é usada para verificar se existem dependências estruturais de identificação encadeada no esquema EER que ainda não foram transformadas para o atual esquema Relacional. Caso o resultado da análise for verdadeiro, os algoritmos classificados como com dependência estrutural encadeadas são executados e, ao final destas execuções, a função *hasChainedDependencies* é novamente testada. Caso o resultado da análise de *hasChainedDependencies* for falso, significa que não existem mais estruturas de identificação encadeada a serem transformadas e, consequentemente, a estrutura de laço chega ao fim. Em seguida, os algoritmos classificados como com dependência de identificação final (i.e, ATRBNN, ATRN e ATRUNN) e associação (i.e, ATRB11, ATRB1N, ATRU11 e ATRU1N) são executados. Por fim, o esquema Relacional é retornado.

4.4 Considerações Finais

Neste Capítulo foram apresentadas as dependências entre os construtores do Modelo EER, a taxonomia da dependência estrutural entre estes construtores, o Catálogo de regras para transformação de esquema EER em esquema Relacional, a classificação destas regras de acordo com a dependência estrutural e o algoritmo para automatizar a execução destas regras. As regras do Catálogo são definidas com foco nas restrições de Participação de Relacionamento, na restrição de Disjunção em Herança e na restrição de Completude em Categoria e Herança. Vale ressaltar que para

garantir estas restrições é necessário o uso de Assertivas ou Gatilhos no esquema Relacional.

No Quadro 11, têm-se as nove características de avaliação definidas no Capítulo 3 para comparar o Catálogo com os trabalhos apresentados ao longo daquele capítulo. Em relação a esses trabalhos, o Catálogo caracteriza-se como uma proposta nova, que visa contribuir como uma solução MDD para preservação das restrições de Participação, Disjunção e Completude na transformação de esquemas EER em esquemas Relacionais. No próximo capítulo são apresentados detalhes técnicos relacionados a especificação do Catálogo em uma ferramenta.

Quadro 11 - Análise Comparativa entre os trabalhos avaliados.

Trabalhos / Características de Avaliação	Embley e Mok	Hainaut	Cuadra *et al.*	Edson Silva
Relacionamento Binário (Participação)	Parcial	Parcial	Sim	Sim
Relacionamento N-ário (Participação)	Parcial	Parcial	Sim	Sim
Relacionamento Unário (Participação)	Parcial	Parcial	Sim	Sim
Relacionamento Identificador (Participação)	Parcial	Não	Não	Sim
Entidade Associativa (Participação)	Não	Não	Não	Sim
Herança (Restrição de Disjunção)	Não	Não	Não	Sim
Herança (Restrição de Completude)	Não	Não	Não	Sim
Categoria (Restrição de Completude)	Não	Não	Não	Sim
Baseado em MDD	Não	Sim	Não	Sim
% Sim	0%	11,1%	33,3	100%
% Não	55,5%	55,5%	66,6	0%
% Parcial	44,4%	33,3%	0%	0%

Fonte: Autor (2015).

5 Avaliação

Este capítulo avalia o Catálogo de regras para transformação do Modelo EER para o Modelo Relacional. O objetivo da avaliação é mostrar, com exemplos, que: 1) As regras são computáveis através de uma ferramenta; e 2) considerando as dependências estruturais e restrições de Participação, Disjunção e Completude, a execução dessas regras produz código SQL-Relacional correspondente ao esquema EER modelado. Para isso, nove cenários de avaliação (i.e., esquemas EER) foram transformados em códigos SQL-Relacional, os quais têm sua sintaxe e semântica conferidas por especialistas. Para alcançar este fim, este capítulo é organizado da seguinte forma: a Seção 5.1 apresenta uma visão arquitetural de como o Catálogo foi implementado na ferramenta EERCASE; na Seção 5.2 são apresentados os cenários de avaliação e na Seção 5.3 são apresentadas as considerações finais do capítulo.

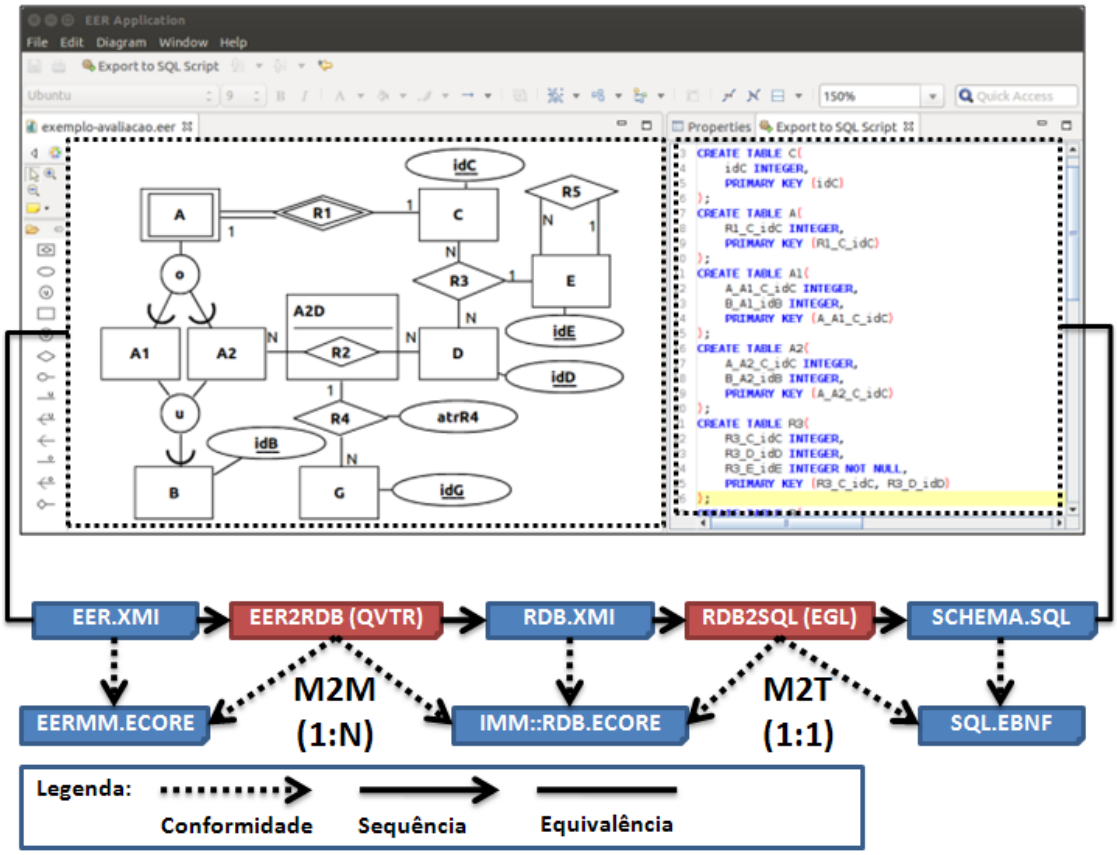
5.1 Implementação do Catálogo na EERCASE

A implementação do Catálogo de regras de transformação é feita na ferramenta EERCASE (*Enchanced Entity Relationship* CASE) (ALVES et al., 2014), pois esta ferramenta segue o paradigma MDD; e se desconhece outra ferramenta que dê suporte a todos os construtores EER segundo a visão de Elmasri e Navathe (ELMASRI; NAVATHE, 2010).

A EERCASE é implementada usando as seguintes tecnologias: 1) *Graphical Modeling Project* (GMP)(GRONBACK, 2009) – arcabouço de tecnologias básicas para construir ferramentas CASE em Java/Eclipse; 2) *Epsilon Framework* – arcabouço de tecnologias que simplifica a construção de ferramentas CASE em GMP (KOLOVOS et al., 2015); 3) *XML Metadata Interchange* (XMI) para armazenamento, manipulação, recuperação e intercâmbio de metadados; 4) linguagem QVT-R para especificar as transformações M2M (OMG, 2011); e 5) linguagem *Epsilon Generation Language* (EGL) para especificar transformações M2T (KOLOVOS et al., 2015). Vale ressaltar que todas estas tecnologias estão em conformidade com

o padrão *Essential Meta Object Facility* (EMOF) (OMG, 2013), cuja implementação no eclipse chama-se ECORE (GRONBACK, 2009).

Figura 32 - Implementação do Catálogo na EERCASE.



Fonte: Autor (2015).

Na Figura 32, ilustra-se como o Catálogo proposto neste trabalho foi implementado na EERCASE. Na parte superior da Figura 32 é mostrada a interface da EERCASE, onde o usuário pode especificar os esquemas EER, executar a transformação e visualizar o código SQL-Relacional gerado. Por sua vez, na parte inferior da Figura 32, pode-se ver que: 1) o esquema EER corresponde a um arquivo EER.XMI, o qual é baseado no metamodelo EERMM.ECORE; 2) a transformação EER2RDB é feita via código QVT-R e baseia-se nos metamodelos EERMM.ECORE e IMM::RDB.ECORE; 3) o arquivo RDB.XMI gerado pela transformação EER2RDB é baseado no metamodelo IMM::RDB.ECORE; 4) a transformação RDB2SQL é feita via código EGL e baseia-se no metamodelo IMM::RDB.ECORE e na gramática SQL.EBNF; e 5) o arquivo SCHEMA.SQL gerado pela transformação RDB2SQL está em conformidade com a gramática SQL.EBNF. Vale ressaltar que a transformação M2M (EER2RDB) implementa os dezessete algoritmos da

Seção 4.2 e, por permitir transformar um elemento EER em mais de um elemento Relacional, tem cardinalidade 1:N. Por sua vez, a transformação M2T (RDB2SQL), por fazer o mapeamento direto entre um elemento relacional e um comando SQL, tem cardinalidade 1:1. Os códigos QVTR e EGL podem ser vistos nos Apêndices A e B, respectivamente. Ressalta-se que na etapa da transformação modelo para texto (i.e., RDB2SQL), foi escolhida a tecnologia Postgres versão 9.3. Todavia, poderia ser escolhido outra tecnologia, como Oracle, MySql ou SQLServer, pois esta etapa é simples de ser especificada, sendo um mapeamento um para um entre os conceitos do Modelo Relacional para o código SQL da tecnologia escolhida.

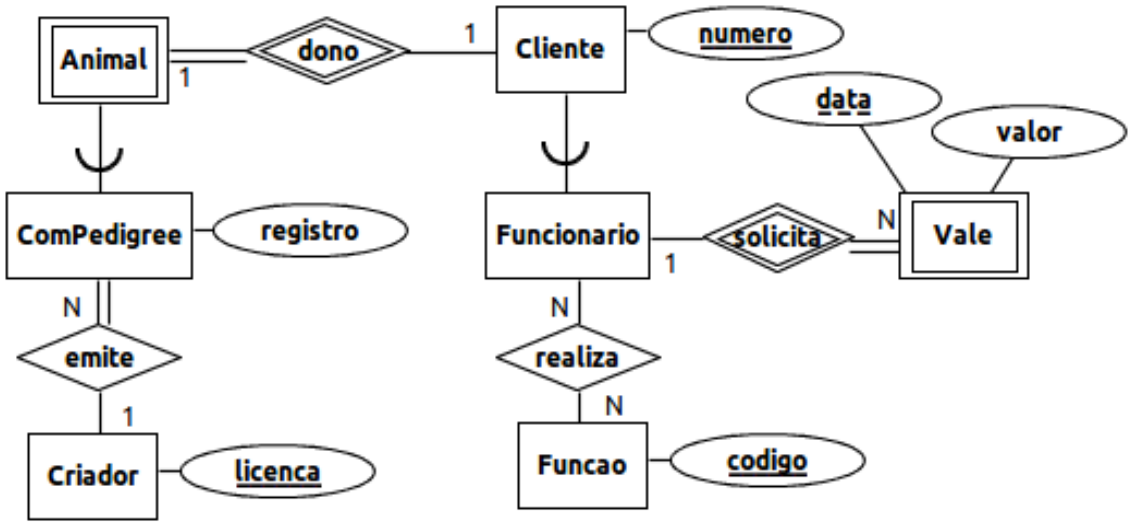
5.2 Avaliação do Trabalho

De modo a verificar a viabilidade deste trabalho, foram consultados seis profissionais especialistas em BD (professores e administradores de BD), os quais avaliaram nove códigos SQL gerados pela ferramenta EERCASE. Ressalta-se que estes cenários são suficientes para mostrar, com detalhes, as dependências estruturais e restrições estruturais não triviais do Modelo EER. O primeiro código gerado é baseado no esquema de motivação deste trabalho (cf. Figura 3). O segundo corresponde a um exemplo simulado que cobre exemplos básicos e não triviais de construções EER. Os demais códigos ilustram esquemas abstratos que cobrem casos não triviais de Participação em Relacionamentos, casos de Disjunção em Herança e casos de Completude em Herança e Categoria. Vale ressaltar que todos os códigos gerados estão sintaticamente válidos, pois foram testados no SGBD PostgreSQL (versão 9.3) e que os Gatilhos e restrições de unicidade são especificados para serem executados apenas ao final da transação. Isto é, os Gatilhos são criados como *Constraint Trigger* (POSTGRESQL, 2015a), e ambos (Gatilhos e restrições de unicidade) são definidos com a cláusula *Initially Deferred* (específico do SGBD Postgres) (POSTGRESQL, 2015b). A seguir, para não ser muito repetitivo e enfadonho, apenas os códigos dos dois primeiros esquemas EER (cenários) são discutidos (os outros sete códigos SQL-Relacional e seus respectivos esquemas EER podem ser consultados no Apêndice C). Na sequência, os comentários dos especialistas são compilados e apresentados.

5.2.1 Cenário 1: Verificação de Dependências Estruturais e Participação no Exemplo de Motivação

Na Figura 33, tem-se um esquema EER que corresponde a uma pequena extensão (i.e., a adição dos Relacionamentos Binários *emite* e *realiza*) feita no exemplo usado na motivação deste trabalho. Esta extensão tem o objetivo de mostrar as dependências estruturais da transformação de Relacionamentos Binários. Por sua vez, código SQL referente a este esquema é apresentado no Quadro 12.

Figura 33 - Verificação de Dependências Estruturais no exemplo de motivação.



Fonte: Autor (2015).

Quadro 12 - Código SQL-Relacional do Cenário 1.

```
-- Criação de Tabelas, Chaves Primárias e demais colunas
CREATE TABLE Cliente(
    numero INTEGER,
    PRIMARY KEY (numero)
);
CREATE TABLE Animal(
    dono_Cliente_numero INTEGER,
    PRIMARY KEY (dono_Cliente_numero)
);
CREATE TABLE ComPedigree(
    registro INTEGER NOT NULL,
    Animal_ComPedigree_Cliente_numero INTEGER,
    emite_Criador_licenca INTEGER NOT NULL,
    PRIMARY KEY (Animal_ComPedigree_Cliente_numero)
);
CREATE TABLE Funcionario(
    Cliente_Funcionario_numero INTEGER,
```

```
    PRIMARY KEY (Cliente_Funcionario_numero)
);
CREATE TABLE Vale(
    valor NUMERIC NOT NULL,
    data TIMESTAMP,
    solicita_Funcionario_numero INTEGER,
    PRIMARY KEY (data, solicita_Funcionario_numero)
);
CREATE TABLE Criador(
    licenca INTEGER,
    PRIMARY KEY (licenca)
);
CREATE TABLE Funcao(
    codigo INTEGER,
    PRIMARY KEY (codigo)
);
CREATE TABLE realiza(
    realiza_Funcionario_numero INTEGER,
    realiza_Funcao_codigo INTEGER,
    PRIMARY KEY (realiza_Funcionario_numero, realiza_Funcao_codigo)
);
--- Criação das restrições de Chaves Estrangeiras
ALTER TABLE Animal ADD CONSTRAINT FK_dono_Cliente FOREIGN KEY
(dono_Cliente_numero) REFERENCES Cliente (numero);
ALTER TABLE ComPedigree ADD CONSTRAINT FK_Animal_ComPedigree FOREIGN
KEY (Animal_ComPedigree_Cliente_numero) REFERENCES Animal
(dono_Cliente_numero);
ALTER TABLE Funcionario ADD CONSTRAINT FK_Cliente_Funcionario FOREIGN
KEY (Cliente_Funcionario_numero) REFERENCES Cliente (numero);
ALTER TABLE Vale ADD CONSTRAINT FK_solicita_Funcionario FOREIGN KEY
(solicita_Funcionario_numero) REFERENCES Funcionario
(Cliente_Funcionario_numero);
ALTER TABLE ComPedigree ADD CONSTRAINT FK_emite_Criador FOREIGN KEY
(emite_Criador_licenca) REFERENCES Criador (licenca);
ALTER TABLE realiza ADD CONSTRAINT FK_realiza_Funcao FOREIGN KEY
(realiza_Funcao_codigo) REFERENCES Funcao (codigo);
ALTER TABLE realiza ADD CONSTRAINT FK_realiza_Funcionario FOREIGN KEY
(realiza_Funcionario_numero) REFERENCES Funcionario
(Cliente_Funcionario_numero);
```

Fonte: Autor (2015).

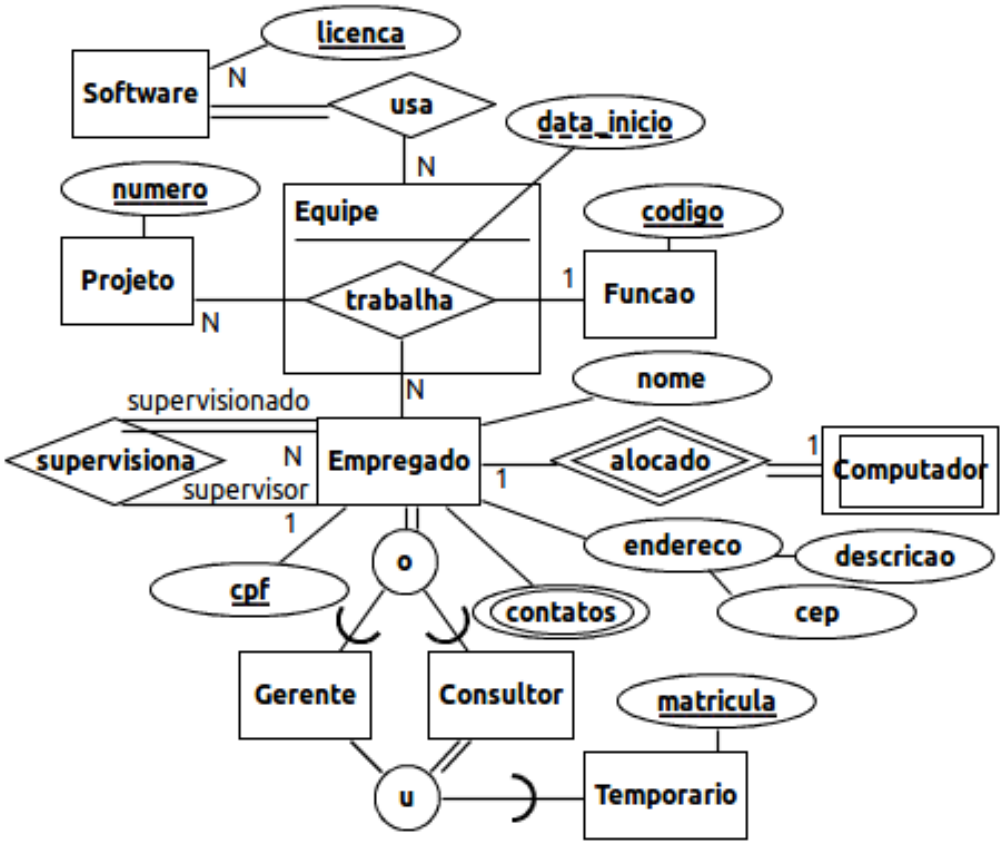
Como pode ser visto no Quadro 12, o código SQL gerado cria as Tabelas e suas Chaves Primárias, para, em seguida, criar as restrições de Chaves Estrangeiras. Desta forma, uma vez que as Chaves Primárias são criadas preservando suas Dependências de Identificação (i.e., Final e Encadeada) e as Chaves Estrangeiras são criadas preservando suas Dependências de Associação (cf. Seção 4.1), o código SQL gerado preserva as Dependências Estruturais que são especificadas no exemplo de motivação. Note que no código SQL gerado não tem Gatilhos para garantir as restrições de Participação Total da Entidade *Animal* no Relacionamento *dono* e da Entidade *Vale* no relacionamento *solicita*, pois, como as Chaves Primárias

destas Entidades também são Chaves Estrangeiras, estas são não anuláveis (i.e., de preenchimento obrigatório). Por fim, ressalta-se que a ordem de criação das Tabelas é influenciada por dois fatores: 1) a ordem que o usuário da EERCASE cria as Entidades e 2) suas Dependências de Identificação. Assim, outras possibilidades de ordem para a criação das tabelas seriam: *Cliente*, *Funcionário*, *Funcao*, *Vale*, *Animal*, *ComPedigree* e *Criador* ou *Cliente*, *Funcionário*, *Animal*, *Funcao*, *Vale*, *Criador* e *ComPedigree*. A seguir, no Cenário 2, são mostrados mais exemplos de construções EER que têm Dependências de Identificação e Associação, bem como restrições de Participação Total que exigem Gatilhos para serem garantidas pelo SGBD.

5.2.2 Cenário 2: Verificação de Dependências Estruturais e Participação em um esquema EER não trivial

O cenário ilustrado na Figura 34 apresenta um pequeno trecho de um esquema EER simulado para uma empresa de engenharia. Vale destacar que o cenário em questão cobre os quatorzes elementos básicos do Modelo EER, a saber: 1) Atributo Simples (i.e., *nome*); 2) Atributo Composto (i.e., *endereço*); 3) Atributo Multivalorado (i.e., *contatos*); 4) Atributo Identificador (i.e., *cpf*); 5) Atributo Discriminador (i.e., *data início*); 6) Entidade Regular (i.e., *Empregado*); 7) Entidade Fraca (i.e., *Computador*); 8) Entidade Associativa (i.e., *Equipe*); 9) Herança (i.e., *Gerente*); 10) Categoria (i.e., *Temporário*); 11) Relacionamento Identificador (i.e., *alocado*); 12) Relacionamento Unário (i.e., *supervisiona*); 13) Relacionamento Binário (i.e., *usa*); e 14) Relacionamento N-ário (i.e., *trabalha*). Além disso, é importante notar que a Entidade *Software* tem Participação Total no Relacionamento *usa* e a Herança é Total, o que exige a especificação de um Gatilho para garantir estas restrições. O código SQL-Relacional deste esquema é mostrado no Quadro 13.

Figura 34 - Verificação de Dependências Estruturais em um esquema EER básico e não trivial.



Fonte: Autor (2015).

Quadro 13 - Código SQL-Relacional do Cenário 2.

```
-- Criação de Tabelas, Chaves Primárias e demais colunas
CREATE TABLE Projeto(
    numero INTEGER,
    PRIMARY KEY (numero)
);
CREATE TABLE Equipe(
    Equipe_Projeto_numero INTEGER,
    Equipe_Empregado_cpf INTEGER,
    Equipe_Funcao_codigo INTEGER NOT NULL,
    data_inicio TIMESTAMP,
    PRIMARY KEY (Equipe_Projeto_numero, Equipe_Empregado_cpf,
data_inicio)
);
CREATE TABLE Funcao(
    codigo INTEGER,
    PRIMARY KEY (codigo)
);
CREATE TABLE Empregado(
    nome VARCHAR(255) NOT NULL,
    cep VARCHAR(255) NOT NULL,
    descricao VARCHAR(255) NOT NULL,
    cpf INTEGER,
    supervisionar_Empregado_cpf INTEGER NOT NULL,
```

```
    PRIMARY KEY (cpf)
);
CREATE TABLE Empregado_contatos(
    contatos VARCHAR(255),
    Empregado_contatos_cpf INTEGER,
    PRIMARY KEY (contatos, Empregado_contatos_cpf)
);
CREATE TABLE Consultor(
    Empregado_Consultor_cpf INTEGER,
    Temporario_Consultor_matricula INTEGER NOT NULL,
    PRIMARY KEY (Empregado_Consultor_cpf)
);
CREATE TABLE Gerente(
    Empregado_Gerente_cpf INTEGER,
    Temporario_Gerente_matricula INTEGER,
    PRIMARY KEY (Empregado_Gerente_cpf)
);
CREATE TABLE Computador(
    alocado_Empregado_cpf INTEGER,
    PRIMARY KEY (alocado_Empregado_cpf)
);
CREATE TABLE Software(
    licenca INTEGER,
    PRIMARY KEY (licenca)
);
CREATE TABLE usa(
    usa_Software_licenca INTEGER,
    usa_Equipe_cpf INTEGER,
    usa_Equipe_numero INTEGER,
    usa_Equipe_data_inicio TIMESTAMP,
    PRIMARY KEY (usa_Software_licenca, usa_Equipe_cpf,
usa_Equipe_numero, usa_Equipe_data_inicio)
);
CREATE TABLE Temporario(
    matricula INTEGER,
    PRIMARY KEY (matricula)
);
--- Criação das restrições de Chaves Estrangeiras
ALTER TABLE Equipe ADD CONSTRAINT FK_Equipe_Projeto FOREIGN KEY
(Equipe_Projeto_numero) REFERENCES Projeto (numero);
ALTER TABLE Empregado_contatos ADD CONSTRAINT FK_Empregado_contatos
FOREIGN KEY (Empregado_contatos_cpf) REFERENCES Empregado (cpf);
ALTER TABLE Equipe ADD CONSTRAINT FK_Equipe_Empregado FOREIGN KEY
(Equipe_Empregado_cpf) REFERENCES Empregado (cpf);
ALTER TABLE Equipe ADD CONSTRAINT FK_Equipe_Funcao FOREIGN KEY
(Equipe_Funcao_codigo) REFERENCES Funcao (codigo);
ALTER TABLE Empregado ADD CONSTRAINT FK_supervisionar_Empregado
FOREIGN KEY (supervisionar_Empregado_cpf) REFERENCES Empregado (cpf);
ALTER TABLE Consultor ADD CONSTRAINT FK_Empregado_Consultor FOREIGN
KEY (Empregado_Consultor_cpf) REFERENCES Empregado (cpf);
ALTER TABLE Gerente ADD CONSTRAINT FK_Empregado_Gerente FOREIGN KEY
(Empregado_Gerente_cpf) REFERENCES Empregado (cpf);
ALTER TABLE Computador ADD CONSTRAINT FK_alocado_Empregado FOREIGN KEY
(alocado_Empregado_cpf) REFERENCES Empregado (cpf);
ALTER TABLE usa ADD CONSTRAINT FK_usa_Software FOREIGN KEY
(usa_Software_licenca) REFERENCES Software (licenca);
ALTER TABLE usa ADD CONSTRAINT FK_usa_Equipe FOREIGN KEY
(usa_Equipe_cpf, usa_Equipe_numero, usa_Equipe_data_inicio) REFERENCES
Equipe (Equipe_Empregado_cpf , Equipe_Projeto_numero, data_inicio );
ALTER TABLE Consultor ADD CONSTRAINT FK_Temporario_Consultor FOREIGN
KEY (Temporario_Consultor_matricula) REFERENCES Temporario
```

```
(matricula);
ALTER TABLE Gerente ADD CONSTRAINT FK_Temporario_Gerente FOREIGN KEY
(Temporario_Gerente_matricula) REFERENCES Temporario (matricula);
-- Criação dos Gatilhos para garantir as participações totais
-----------------------------------------------------------------------
/*O gatilho TRG_usa_usa garante a participação total de Software em
usa depois de excluir um registro de usa ou alterar o valor da chave
estrangeira de Software em usa. Lança uma exceção quando não houver
registro da chave estrangeira de Software em usa */
CREATE FUNCTION FUNCTION_TRG_usa_usa()
RETURNS TRIGGER AS $BODY$
    DECLARE countSoftware INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countSoftware
        FROM usa
        WHERE usa.usa_Software_licenca = OLD.usa_Software_licenca;
        IF countSoftware = 0 THEN
            RAISE EXCEPTION 'TRG_usa_usa';
        END IF;
        IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_usa_usa AFTER DELETE OR UPDATE OF
usa_Software_licenca ON usa INITIALLY DEFERRED FOR EACH ROW EXECUTE
PROCEDURE FUNCTION_TRG_usa_usa();
-----------------------------------------------------------------------
/*O gatilho TRG_usa_Software garante a participação total de Software
em usa depois de inserir um registro em Software. Lança uma exceção
quando não houver registro da chave estrangeira de Software em usa */
CREATE FUNCTION FUNCTION_TRG_usa_Software()
RETURNS TRIGGER AS $BODY$
    DECLARE countSoftware INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countSoftware
        FROM usa
        WHERE usa.usa_Software_licenca = NEW.licenca;
        IF countSoftware = 0 THEN
            RAISE EXCEPTION 'TRG_usa_Software';
        END IF;
        RETURN NEW;
    END
$BODY$LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_usa_Software AFTER INSERT ON Software
INITIALLY DEFERRED
FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_usa_Software();
-----------------------------------------------------------------------
/*O gatilho TRG_Empregado_Empregado garante a completude total de
Empregado na herança depois de inserir um registro em Empregado. Lança
uma exceção quando não houver registro da chave estrangeira de
Empregado em Gerente ou Consultor */
CREATE FUNCTION FUNCTION_TRG_Empregado_Empregado()
RETURNS TRIGGER AS $BODY$
    DECLARE countConsultor INTEGER;
    DECLARE countGerente INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countConsultor
        FROM Consultor
```

```
        WHERE Consultor.Empregado_Consultor_cpf = NEW.cpf;
        SELECT COUNT(*) INTO countGerente
        FROM Gerente
        WHERE Gerente.Empregado_Gerente_cpf = NEW.cpf;
        IF countConsultor + countGerente = 0 THEN
            RAISE EXCEPTION 'TRG_Empregado_Empregado';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_Empregado_Empregado AFTER INSERT ON
Empregado INITIALLY DEFERRED
FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_Empregado_Empregado();
-----------------------------------------------------------------------
/*O gatilho TRG_Empregado_Gerente garante a completude total de
Empregado na herança depois de excluir um registro de Gerente ou
alterar o valor da chave estrangeira de Empregado em Gerente. Lança
uma exceção quando não houver registro da chave estrangeira de
Empregado em Gerente */
CREATE FUNCTION FUNCTION_TRG_Empregado_Gerente()
RETURNS TRIGGER AS $BODY$
    DECLARE countConsultor INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countConsultor
        FROM Consultor
        WHERE Consultor.Empregado_Consultor_cpf =
OLD.Empregado_Gerente_cpf;
        IF countConsultor = 0 THEN
            RAISE EXCEPTION 'TRG_Empregado_Gerente';
        END IF;
       IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_Empregado_Gerente AFTER DELETE OR UPDATE
OF Empregado_Gerente_cpf ON Gerente INITIALLY DEFERRED  FOR EACH ROW
EXECUTE PROCEDURE FUNCTION_TRG_Empregado_Gerente();
-----------------------------------------------------------------------
/* O gatilho TRG_Empregado_Consultor garante a completude total de
Empregado na herança depois de excluir um registro de Consultor ou
alterar o valor da chave estrangeira de Empregado em Consultor. Lança
uma exceção quando não houver registro da chave estrangeira de
Empregado em Consultor */
CREATE FUNCTION FUNCTION_TRG_Empregado_Consultor()
RETURNS TRIGGER AS $BODY$
    DECLARE countGerente INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countGerente
        FROM Gerente
        WHERE Gerente.Empregado_Gerente_cpf =
OLD.Empregado_Consultor_cpf;
        IF countGerente = 0 THEN
            RAISE EXCEPTION 'TRG_Empregado_Consultor';
        END IF;
       IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
```

```
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_Empregado_Consultor AFTER DELETE OR
UPDATE OF Empregado_Consultor_cpf ON Consultor INITIALLY DEFERRED FOR
EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_Empregado_Consultor();
```

Fonte: Autor (2015).

Seguindo o mesmo raciocínio do cenário anterior, o código SQL apresentado no Quadro 13, cria primeiro as Tabelas e suas Chaves Primárias, para depois criar as restrições de Chaves Estrangeiras. Ou seja, pelas mesmas razões explicadas no cenário anterior, o código SQL gerado preserva as Dependências Estruturais especificadas no esquema EER do cenário 2. Ademais, destaca-se que: 1) a Participação Total da Entidade *Software* no Relacionamento *usa* exige a especificação de dois Gatilhos (i.e., *TRG_usa_Software* e *TRG_usa_usa*); e 2) a Herança Total da Entidade *Empregado* exige a criação de três Gatilhos (*TRG_Empregado_Empregado*, *TRG_Empregado_Gerente* e *TRG_Empregado_Consultor*). O Gatilho *TRG_usa_Software* verifica se depois de inserir um registro em *Software*, este registro também é referenciado em *usa.* O Gatilho *TRG_usa_usa* verifica se depois de excluir um registro de *usa* ou atualizar o valor da Chave Estrangeira de *Software* em *usa*, ainda existe ao menos uma referência da Chave Estrangeira de *Software* em *usa* que corresponda ao registro excluído ou atualizado. O Gatilho *TRG_Empregado_Empregado* garante a Completude Total de *Empregado* na herança depois de inserir um registro em *Empregado*. Lança uma exceção quando não houver registro da chave estrangeira de *Empregado* em *Gerente* ou *Consultor*. Os Gatilho *TRG_Empregado_Gerente* garante a Completude Total de *Empregado* na herança depois de excluir um registro de *Gerente* ou alterar o valor da chave estrangeira de *Empregado* em *Gerente*. Lança uma exceção quando não houver registro da chave estrangeira de *Empregado* em *Gerente* e o Gatilho *TRG_Empregado_Consultor* garante a Completude Total de *Empregado* na herança depois de excluir um registro de *Consultor* ou alterar o valor da chave estrangeira de *Empregado* em *Consultor*. Lança uma exceção quando não houver registro da chave estrangeira de *Empregado* em *Consultor*.

5.2.3 Avaliação feita pelos especialistas

Dada a complexidade do trabalho e o pouco tempo para avaliá-lo (apenas a semana de 02/02/2015 a 06/02/2015), optou-se por fazer uma avaliação informal e descritiva a partir da condução de uma entrevista semi-estruturada. Foram entrevistados seis especialistas de BD (dois professores, dois profissionais de mercado que não são docentes e dois profissionais do mercado que também são docentes). Os esquemas EER e seus códigos SQL foram discutidos com cada um dos seis especialistas e as entrevistas foram conduzidas a partir das dezessete perguntas a seguir. Estas perguntas foram desenvolvidas para verificar se: 1) os cenários abrangem todos os construtores do Modelo EER; e 2) se os construtores foram transformados de forma correta. Vale ressaltar que o tempo médio para realizar esta atividade com cada entrevistado foi de três horas e trinta minutos.

Quadro 14 - Perguntas da Entrevista.

Perguntas
1. Os esquemas EER apresentados nos cenários **1 e 2 são complexos** para ser transformado em SQL?
2. Os códigos SQL gerados para os cenários **1 e 2 preservam a semântica das cardinalidades e participações dos relacionamentos** especificados nos seus esquemas EER?
3. Os códigos SQL gerados para os cenários **1 e 2 preservam a semântica das completudes das heranças** especificadas nos seus esquemas EER?
4. O **cenário 3** cobre ao menos um caso complexo para as restrições de **cardinalidade e participação** em **relacionamentos binários**?
5. O código SQL que é gerado para o **cenário 3 preserva a semântica das restrições de cardinalidade e participação** especificadas no esquema EER?
6. O **cenário 4** cobre ao menos um caso complexo para as restrições de **cardinalidade e participação** em um **relacionamento N-ário**?
7. O código SQL que é gerado para o **cenário 4 preserva a semântica das restrições de cardinalidade e participação** especificadas no esquema EER?
8. O **cenário 5** cobre ao menos um caso complexo para as restrições **de cardinalidade e participação** em **relacionamentos identificadores**?
9. O código SQL que é gerado para o **cenário 5 preserva a semântica das restrições de cardinalidade e participação** especificadas no esquema

EER?
10. O **cenário 6** cobre ao menos um caso complexo para as restrições **de cardinalidade e participação** em um **relacionamento unário**?
11. O código SQL que é gerado para o **cenário 6 preserva a semântica das restrições de cardinalidade e participação** especificadas no esquema EER?
12. O **cenário 7** cobre ao menos um caso complexo para as restrições **de cardinalidade e participação** usando **entidades associativas**?
13. O código SQL que é gerado para o **cenário 7 preserva a semântica das restrições de cardinalidade e participação** especificadas no esquema EER?
14. O **cenário 8** cobre ao menos um caso complexo para as restrições de **completude e disjunção** em **heranças**?
15. O código SQL que é gerado para o **cenário 8 preserva a semântica das restrições de completude e disjunção** especificadas no esquema EER?
16. O **cenário 9** cobre ao menos um caso complexo para a restrição de **completude** em **categoria**?
17. O código SQL que é gerado para o **cenário 9 preserva a semântica da restrição de completude** especificada no esquema EER?

Fonte: Autor (2015).

As avaliações feitas pelos entrevistados podem ser consolidadas em dois pontos: 1) as construções exemplificadas nos esquemas EER cobrem todos os construtores EER e são não triviais. Contudo, também foi dito que algumas construções não são frequentes na prática, pois são demasiadamente complexas e restritivas; e 2) os códigos SQL gerados capturam corretamente as restrições semânticas especificadas em seus esquemas EER. Porém, também foi comentado que, apesar dos códigos SQL serem uma evidência de que as transformações cobrem casos não triviais de forma completa e correta, ainda falta realizar um trabalho mais rigoroso (baseado em verificação formal) para provar que as transformações atendem às propriedades de corretude e completude.

5.3 Considerações Finais

Neste Capítulo, o Catalogo de regras de transformação proposto foi avaliado. Esta avaliação foi feita em dois passos, a saber: 1) o primeiro passo foi a implementação do Catalogo na ferramenta EERCASE, visando mostrar

que as regras do Catálogo são computáveis; e 2) o segundo passo consistiu na avaliação, por especialistas em BD, dos códigos SQL-Relacional gerados a partir de cenários com esquemas EER predefinidos. Com esta avaliação constatou-se que o Catálogo cobre casos complexos de transformação, no entanto, não é possível garantir que o Catálogo é completo e correto. No próximo capítulo, são apresentadas as conclusões, contribuições e indicações de trabalhos futuros.

6 Conclusão

Este capítulo apresenta as considerações finais sobre o trabalho desenvolvido, suas principais contribuições, limitações e sugestões de trabalhos futuros.

6.1 Considerações Finais

O principal objetivo deste trabalho foi garantir que o esquema fisíco implementado reflita a semântica especificada no esquema conceitual, evitando que erros sejam cometidos pelo projetista do projeto de BD. Este objetivo foi alcançado com a definição de um Catálogo de regras para transformação automática de esquemas EER em esquemas Relacionais e este em código SQL-Relacional, considerando as restrições estruturais não triviais e as dependências estruturais do Modelo EER.

O trabalho contasta que na transformação de esquemas EER para esquemas Relacionais existe um consenso quanto às regras para transformar os conceitos básicos do EER (i.e., Entidades, Relacionamentos e Atributos). No entanto, existem restrições não triviais do Modelo EER (i.e., Participação, Disjunção e Completude) que não possuem regras de transformação claramente definidas. Além disso, como pode ser visto na Figura 3, a transformação de um esquema EER em um esquema Relacional não é um processo sequencial, pois existem construções que têm dependências estruturais que devem ser executadas primeiro que outras. Estas limitações não são consideradas na definição das regras para transformação encontradas nos livros clássicos para BD e não se encontrou um trabalho que as explorasse por completo (cf. Seção 3). Ressalta-se que a definição desta ordem é importante para executar estas regras automaticamente em um ambiente computacional.

A solução proposta solucionou o problema das restrições estruturais não triviais com a especificação do Catálogo de regras para transformação de esquemas EER em Relacionais (cf. Seção 4.2). Além disso, elucidou-se o

problema das dependências estruturais com a definição (1) das dependências estruturais dos construtores do Modelo EER (com base no metamodelo EERMM) (cf. Quadro 8), (2) da taxonomia das dependências estruturais (cf. Figura 11) e (3) do algoritmo para execução das regras do Catálogo (cf. Algoritmo 18), que é baseado na taxonomia das dependências estruturais proposta.

A avaliação deste trabalho constou inicialmente da implementação do Catálogo na ferramenta EERCASE, no qual se mostra que as regras de transformação apresentadas neste trabalho são automatizáveis. Em seguida, especialistas em BD verificam a corretude da sintaxe e semântica de códigos SQL gerados a partir de esquemas EER modelados na ferramenta EERCASE. Ressalta-se que a implementação do Catálogo no processo de transformação MDD da ferramenta EERCASE se da na etapa modelo para modelo (M2M), pois esta é mais complexa de ser realizada, uma vez que um conceito do Modelo EER pode ser mapeado como vários conceitos do Modelo Relacional. Além disso, na etapa da transformação modelo para texto (M2T), foi escolhido à tecnologia Postgres versão 9.3. Todavia, poderia ser escolhido outra tecnologia, como Oracle, MySql ou SQLServer, pois esta etapa é simples de ser especificada, sendo um mapeamento um para um entre os conceitos do Modelo Relacional para o código SQL da tecnologia escolhida.

6.2 Contribuições

Após a conclusão deste trabalho, podem ser citadas como principais contribuições:

- A identificação das dependências estruturais entre os construtores do Modelo EER (cf. Quadro 8). Nesta análise, constatou-se que existem apenas dez combinações distintas que têm dependências, portanto, necessitam que a ordem de execução das transformações dos construtores envolvidos seja analisada e definida. As demais combinações não exigem esta análise, pois não têm dependência entre os construtores envolvidos ou são proibidas pelo metamodelo EERMM. Ou seja, quando permitidas, podem ser executadas em qualquer ordem.

- A taxonomia das dependências estruturais entre os construtores do Modelo EER (cf. Figura 11). Esta taxonomia foi criada com base no estudo das dependências estruturais encontradas no Quadro 8 e consiste em uma classificação para os tipos de dependências estruturais (i.e, identificação encadeada, identificação final e associação). Sendo essa classificação importante para decidir a ordem da execução das regras de transformação.

- O Catálogo de regras para transformação de esquemas EER em esquemas Relacionais e este em código SQL. Este Catálogo é especificado de forma algorítmica, com base nos metamodelos EERMM e IMM Relacional (cf. Seção 4.2). Destaca-se que o Catálogo foi especificado com foco na preservação das restrições não triviais.

- O Algoritmo para definir a ordem de execução na transformação de esquema EER em esquema Relacional (cf. Seção 4.3). Este algoritmo define a lógica que permite executar as regras de transformações em uma ordem correta com base nas dependências estruturais discutidas na Seção 4.1.

Além das contribuições principais, também podem ser citadas como contribuições secundárias do trabalho:

- A especificação das regras do Catálogo na linguagem QVT-R. Pois não se encontrou nenhum outro trabalho que utilize esta linguagem em um projeto real.

- A especificação da transformação M2T do Modelo Relacional (com base no metamodelo IMM Relacional) para código SQL compatível com o SGBD Postgres versão 9.3;

- A implementação das regras do Catálogo no processo de transformação MDD da ferramenta EERCASE;

- A adaptação do metamodelo EERMM. Pois sem a adição da propriedade “*chosenLink*” na classe “*RelationshipLink*” as transformações seriam ambíguas;

- A criação da visão de implementação do metamodelo IMM Relacional (cf. Figura 7). Pois a especificação do mesmo não contemplava o padrão ECORE, necessário para implementação da transformação na plataforma Java/Eclispse.

6.3 Publicações

Além das contribuições citadas na seção anterior, foram publicados trabalhos relacionadas as linhas de pesquisa da dissertação. Estas são importante como parte da experiência de uma pesquisa em nível de mestrado.

ALVES, E. FRANCO, N. NASCIMENTO, A. FIDALGO, R. EERCASE: Uma Ferramenta para Apoiar o Estudo do Projeto Conceitual de Banco de Dados. **Anais dos Workshops do CBIE 2014**, v. 1, n. 1, p. 98–105, 2014.

FIDALGO, R. ALVES, E. ESPAÑA, S. CASTRO, J. PASTOR, O. Metamodeling the Enhanced Entity-Relationship Model. **Journal of Information and Data Management**, v. 4, n. 3, p. 406–420, 2013.

6.4 Limitações

Foram identificadas algumas limitações na solução proposta. Que incluem:

- As estratégias usadas para especificação das regras de transformação foram as de adição de Coluna ou criação de Tabela. No entanto, em se tratando de construtores como o Relacionamento Binário com Cardinalidade 1:1, recomenda-se a estratégia de Fusão de Tabelas. Esta limitação se deve ao fato da complexidade dos algoritmos, que criariam novas variáveis deficies de serem gerenciadas;

- A escolha da estratégia para transformação (i.e., adição de Colunas, criação ou fusão de Tabelas) é uma decisão de projeto. No entanto, neste trabalho não é permitido ao usuário a escolha do tipo de estratégia; pois sempre é executada a estratégia que foi julgada mais eficiente para a maioria dos casos. Desta forma, uma limitação deste trabalho é não permitir que a estratégia da transformação seja especificada pelo usuário da ferramenta;

- Não foi realizada a avaliação do Catálogo em um caso real.

6.5 Trabalhos Futuros

Trabalhos futuros que podem evoluir ou estender a proposta apresentada nesta dissertação:

- Utilizar uma abordagem formal para verificar as propriedades de completude e corretude das regras de transformações do Catálogo (i.e., utilizar simulação ou verificação de modelos);

- Permitir ao projetista do BD especificar detalhes sobre o comportamento da transformação (e.g., fusão de Tabela, adição de Coluna ou criação de Tabela). Esta melhoria implica na extensão do EERMM e, consequentemente, na atualização do Catálogo.

- Realizar experimentos com projetos reais de projeto de BD. Pois, com os resultados desta dissertação, é possível criar um conjunto de boas práticas sobre modelagens que são válidas, contudo não são recomendadas em um projeto real.

Referência

ACM/AIS. **Curriculum guidelines for undergraduate degree programs in information technology**. Disponível em: <http://www.acm.org/education/curricula/IS 2010 ACM final.pdf>. Acesso em: 15 fev. 2015.

ACM/IEEE. **Computer Science Curriculum 2008: an interim revision of CS 2001**. Disponível em: <http://www.acm.org/education/curricula/ComputerScience2008.pdf>. Acesso em: 15 fev. 2015.

ALVES, E.; FRANCO, N.; Nascimento, A.; FIDALGO, R. EERCASE: Uma Ferramenta para Apoiar o Estudo do Projeto Conceitual de Banco de Dados. **Anais dos Workshops do CBIE 2014**, v. 1, n. 1, p. 98–105, 2014.

AMALFI, M.; ARTALE, A.; CALI, A.; PROVETTI, A. Generating Preview Instances for the Face Validation of Entity-Relationship Schemata: The Acyclic Case. In: YU, J.; KIM, M.; UNLAND, R. (Eds.). . **Database Systems for Advanced Applications**. Lecture Notes in Computer Science. [s.l.] Springer Berlin Heidelberg, 2011. v. 6588p. 225–234.

BADIA, A.; LEMIRE, D. A call to arms: revisiting database design. **ACM SIGMOD Record**, p. 61–69, 2011.

BAVOTA, G.; GRAVINO, C.; OLIVETO, R.; DE LUCIA, A.; TORTORA, G.; GENERO, M.; CRUZ-LEMUS, J. Identifying the Weaknesses of UML Class Diagrams during Data Model Comprehension. In: WHITTLE, J.; CLARK, T.; KÜHNE, T. (Eds.). . **Model Driven Engineering Languages and Systems**. Lecture Notes in Computer Science. [s.l.] Springer Berlin Heidelberg, 2011. v. 6981p. 168–182.

BIEHL, M. Literature study on model transformations. **Royal Institute of Technology**, v. 1, n. 1, p. 1–28, 2010.

BOLLATI, V.; ATZENI, P.; MARCOS, E.; VARA, J. Model management systems vs. model driven engineering: a case study. **Proceedings of the 27th Annual ACM Symposium on Applied Computing**, p. 865–872, 2012.

BRAMBILLA, M.; CABOT, J.; WIMMER, M. **Model-Driven Software Engineering in Practice**. 1. ed. San Rafael, CA, USA: Morgan&Claypool, 2012. v. 1

CHEN, P. The entity-relationship model—toward a unified view of data. **ACM Transactions on Database Systems (TODS)**, v. 1, n. 1, p. 9–36, 1976.

CHO, H.; GRAY, J.; SYRIANI, E. Creating visual domain-specific modeling languages from end-user demonstration. **Modeling in Software Engineering (MISE)**, v. 1, n. 1, p. 22–28, 2012.

CODD, E. F. A relational model of data for large shared data banks. **Comunications of the ACM**, v. 15, n. 3, p. 162–6, 1970.

CUADRA, D.; NIETO, C.; MARTINEZ, P.; CASTRO, E.; VELASCO, M. Preserving Relationship Cardinality Constraints in Relational Schemata. In: RIVERO, J. H. D. AND L. C. (Ed.). . **Database Integrity: Challenges and Solutions**. 1. ed. Hershey, PA: Idea Group Publishing, 2002. v. 2002p. 66 – 112.

CUADRA, D.; MARTÍNEZ, P.; CASTRO, E.; AL-JUMAILY, H. Guidelines for representing complex cardinality constraints in binary and ternary relationships. **Software & Systems Modeling**, v. 12, n. 4, p. 871–889, 24 fev. 2013.

CZARNECKI, K.; HELSEN, S. Classification of model transformation approaches. **Techniques in the Context of the Model**, p. 1–17, 2003.

DATE, C. J. **An Introduction to Database Systems**. 8. ed. Boston, USA: Addison-Wesley, 2003.

DAVIES, I.; GREEN, P.; ROSEMANN, M.; INDULSKA, M.; GALLO, S. How do practitioners use conceptual modeling in practice? **Data & Knowledge Engineering**, v. 58, n. 3, p. 358–380, set. 2006.

DE LUCIA, A.; GRAVINO, C.; OLIVETO, R.; TORTORA, G. Data Model Comprehension: An Empirical Comparison of ER and UML Class Diagrams. **2008 16th IEEE International Conference on Program Comprehension**, p. 93–102, jun. 2008.

DE LUCIA, A.; GRAVINO, C.; OLIVETO, R.; TORTORA, G. An experimental comparison of ER and UML class diagrams for data modelling. **Empirical Software Engineering**, v. 15, n. 5, p. 455–492, 11 dez. 2009.

DULLEA, J.; SONG, I.-Y.; LAMPROU, I. An analysis of structural validity in entity-relationship modeling. **Data & Knowledge Engineering**, v. 47, n. 2, p. 167–205, nov. 2003.

ELMASRI, R.; NAVATHE, S. **Fundamentals of Database Systems**. 6th. ed. USA: Addison-Wesley Publishing Company, 2010.

EMBLEY, D.; MOK, W. Mapping conceptual models to database schemas. **Handbook of Conceptual Modeling**, 2011.

FIDALGO, R.; MARANHÃO, E.; ESPAÑA, S.; CASTRO, J.; PASTOR, O. EERMM: A Metamodel for the Enhanced Entity-Relationship Model. **Conceptual Modeling**, v. 31, n. 1, p. 515–524, 2012.

FIDALGO, R.; ALVES, E.; ESPAÑA, S.; CASTRO, J.; PASTOR, O. Metamodeling the Enhanced Entity-Relationship Model. **Journal of Information and Data Management**, v. 4, n. 3, p. 406–420, 2013.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. **Database Systems: The Complete Book**. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008.

GRONBACK, R. **Eclipse modeling project: a domain-specific language (DSL) toolkit**. 1. ed. San Francisco: Addison-Wesley, 2009.

HAINAUT, J. The transformational approach to database engineering. **Transformational Techniques in Software Engineering**, 2006.

HARTMANN, S. Reasoning about participation constraints and Chen's constraints. **Proceedings of the 14th Australasian database Conference**, 2003.

HEUSER, C. A. **Projeto de banco de dados**. 4. ed. Porto Alegre: Sagra Luzzatto, 1998.

HUBER, P. The model transformation language jungle-an evaluation and extension of existing approaches. **Master's thesis, Business Informatics Group, TU Wien**, 2008.

ISO. **ISO/IEC 9075-14:2003**. Disponível em: <http://www.iso.org/iso/catalogue_detail .htm?csnumber=35341>. Acesso em: 15 fev. 2015.

JILANI, A.; USMAN, M.; HALIM, Z. Model Transformations in Model Driven Architecture. **Universal Journal of Computer Science**, v. 1, n. 1, p. 50–54, 2010.

KELLY, S.; TOLVANEN, J. **Domain-Specific Modeling: Enabling Full Code Generation**. 1. ed. Hoboken, New Jersey: Wiley-IEEE Computer Society, 2007.

KLEPPE, A. G.; WARMER, J. B.; BAST, W. **MDA Explained: The Model Driven Architecture: Practice and Promise**. Boston, USA: Addison-Wesley, 2003.

KOLOVOS, D.; ROSE, L.; GARCÍA-DOMÍNGUEZ, A.; PAIGE, R. **The epsilon book**. Disponível em: <http://www.eclipse.org/epsilon/doc/book/>. Acesso em: 15 fev. 2015.

KROENKE, D. M.; AUER, D. J. **Database Processing: Fundamentals, Design, and Implementation**. 13. ed. New Jersey: Prentice Hall, 2013.

MENS, T.; GORP, P. VAN. A taxonomy of model transformation. **Electronic Notes in Theoretical Computer Science**, p. 1–10, 2006.

MOODY, D.; SHANKS, G. Improving the quality of data models: empirical validation of a quality management framework. **Information systems**, v. 28, p. 619–650, 2003.

MOTTA, G.; PIGNATELLI, G. From Strategic to Conceptual Information Modelling: A Method and a Case Study. In: D'ATRI, A. et al. (Eds.). . **Information Technology and Innovation Trends in Organizations**. Physica-Verlag HD, 2011. p. 179–186.

OMG. **Information Management Metamodel (IMM) Specification**. Disponível em: <http://www.omgwiki.org/imm/lib/exe/fetch.php?id=welcome_to_imm&cache=cache&media=submission_-_ii.pdf>. Acesso em: 15 fev. 2015.

OMG. **Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification**. Needham, MA. Disponível em: <http://www.omg.org/spec/QVT/1.1>.

OMG. **Essential Meta-Object Facility (EMOF)**. Disponível em: <http://www.omg.org/spec/mof/2.4.1/pdf>.

PARENT, C.; SPACCAPIETRA, S.; ZIMÁNYI, E. Spatio-temporal conceptual models: data structures+ space+ time. **Proceedings of the 7th ACM international symposium on Advances in geographic information systems**, 1999.

POSTGRESQL. **PostgreSQL: Documentation: 9.0: CREATE CONSTRAINT TRIGGER**. Disponível em: <http://www.postgresql.org/docs/9.0/static/sql-createconstraint.html> Acesso em: 11 fev. 2015a.

POSTGRESQL. **PostgreSQL: Documentation: 9.0: CREATE TABLE**. Disponível em: <http://www.postgresql.org/docs/9.0/static/sql-createtable.html>. Acesso em: 11 fev. 2015b.

RAMAKRISHNAN, R.; GEHRKE, J. **Database management systems**. 3. ed. New York: McGraw-Hill, 2002.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. **Unified Modeling Language Reference Manual, The (2Nd Edition)**. Pearson Higher Education, 2004.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Database System Concepts**. 6. ed. New York: McGraw-Hill, 2010.

SONG, I.; EVANS, M.; PARK, E. A comparative analysis of entity-relationship diagrams. **Journal of Computer and Software**, v. 3, n. 4, p. 427–459, 1995.

SONG, I.-Y.; CHEN, P. Entity Relationship Model. In: LIU, L.; ÖZSU, M. T. (Eds.). . **Encyclopedia of Database Systems**. Springer US, 2009. p. 1003–1009.

TEOREY, T.; LIGHTSTONE, S.; NADEAU, T.; JAGADISH, H. **Database Modeling and Design: Logical Design**. 5. ed. Massachusetts: Morgan Kaufmann, 2011.

TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. A. G. DO. Introdução à Engenharia de Software Experimental. **Relatório Técnico RT-ES 590/02**, p. 52, 2002.

Apêndice A

Este apêndice contém um fragmento do código QVT utilizado na transformação M2M, do Modelo EER (EERMM) para Modelo Relacional (IMM) do Capítulo 5. O código QVT completo pode ser acessado em http://cin.ufpe.br/~eas4/eer2rdb.qvt.

Quadro 15 - Código QVT-R.

```
transformation eerTorelational(eer:eer, rdbm:relational){
   key relational::DataType{name};
   key relational::Column{name, table};
   key relational::ForeignKey{name};
   key relational::UniqueConstraint{name};
   key relational::Trigger{name};
   key relational::BaseTable{name};

top relation schema_EER_to_schema_REL {
  checkonly domain eer schema_EER : eer::Schema {};
  enforce domain rdbm schema_Rel : relational::Schema {
    name = 'fromEER'
  };
}

top relation entity_to_table {
  e_name : String;
  checkonly domain eer e: eer::Entity{
    name = e_name,
    schema = schema_EER : eer::Schema{}
  };
  enforce domain rdbm t : relational::BaseTable {
    name = e_name ,
    schema = schema_REL : relational::Schema{}
  };
  when{
    schema_EER_to_schema_REL(schema_EER, schema_REL);
    e.isWeak = false;
    e.inheritanceSL->size()=0;
    e.directInheritanceLinkTarget->size()=0;
  }
  where{
    element_attribute_to_column('', e, t);
    element_attribute_compound_to_column('', e , t);
    element_attribute_to_pk('', e.name, e , t);
    element_attribute_compound_to_pk('', e.name, e , t);
    element_attribute_to_table(e.name, e.name, e, t, e);
    element_attribute_compound_to_table(e.name, e.name, e, t, e);
  }
}

top relation weak_entity_to_table{
  t_l : relational::BaseTable;
  r_l_l : eer::RelationshipLink;
  e_l : eer::EntityType ;
```

```
 entity_right : eer::EntityType ;
 checkonly domain eer w_e: eer::Entity {
  isWeak = true,
  relationshipLink = r_l_r : eer::RelationshipLink{
   isIdentifier = true,
   target = r : eer::Relationship {}
  }
 };
 enforce domain rdbm t : relational::BaseTable {
  name = weak_entity.name
 };
 when{
  r.associativeEntity.oclIsUndefined();  r.relationshipLink->size() = 2;
  r_l_l = r.relationshipLink->select(x : eer::RelationshipLink | x.isIdentifier = false)->at(1);
  r_l_l <> r_l_r;  e_l = r_l_l.source;
  entity_right = r_l_r.source;
  entity_to_table(e_l, t_l) or weak_entity_to_table(e_l, t_l) or
  direct_inheritance_to_table(e_l, t_l) or specialized_inheritance_to_table(e_l, t_l) or
  associative_entity_1_1_to_table(e_l, t_l) or associative_entity_1_N_to_table(e_l, t_l) or
  associative_entity_N_N_to_table(e_l, t_l) or associative_entity_nary_to_table(e_l, t_l);
 }
 where{
  element_attribute_to_column('', w_e, t);
  element_attribute_compound_to_column('', w_e, t);
  element_attribute_to_pk('', w_e.name, w_e , t);
  element_attribute_compound_to_pk('', w_e.name, w_e , t);
  element_attribute_to_table(w_e.name, w_e.name, w_e, t, w_e);
  element_attribute_compound_to_table(w_e.name, w_e.name,w_e,t,w_e);
  element_attribute_to_column(r.name+'_', r, t);
  element_attribute_compound_to_column(r.name+'_', r , t);
  element_attribute_to_table(w_e.name, w_e.name, r, table, w_e);
  element_attribute_compound_to_table(w_e.name, w_e.name, r,t, w_e);
  element_attribute_to_pk(r.name+'_'+e_l.name+'_', w_e.name,e_l ,t);
  element_attribute_to_fk(r.name+'_'+e_l.name, e_l.name, 'false', t, e_l, t_l);
  element_attribute_compound_to_fk(r.name+'_'+e_l.name, e_l.name, 'false', t, e_l, t_l);
  weak_entity_to_pk(r.name+'_'+e_l.name, w_e.name, e_l , t);
  weak_entity_to_fk(r.name+'_'+e_l.name, w_e name,'false', t_l , e_l , t);
  direct_inheritance_to_pk(r.name+'_'+e_l.name, w_e.name, e_l, t);
  direct_inheritance_to_fk(r.name+'_'+e_l.name, w_e.name, 'false', t_l ,e_l, t);
  specialized_inheritance_to_pk(r.name+'_'+e_l.name, w_e.name, e_l , t);
  specialized_inheritance_to_fk(r.name+'_'+e_l.name, w_e.name,'false', t_l , e_l , t);
  associative_entity_1_1_to_pk(r.name+'_'+e_l.name, w_e.name, e_l , t);
  associative_entity_1_1_to_fk(r.name+'_'+e_l.name, w_e.name,'false', t_l , e_l , t);
  associative_entity_1_N_to_pk(r.name+'_'+e_l.name, w_e.name, e_l , t);
  associative_entity_1_N_to_fk(r.name+'_'+e_l.name, w_e.name,'false', t_l , e_l , t);
  associative_entity_N_N_to_pk(r.name+'_'+e_l.name, w_e.name, e_l , t);
  associative_entity_N_N_to_fk(r.name+'_'+e_l.name, w_e.name,'false', t_l , e_l , t);
  if r_l_l.participation = eer::CompletenessType::TOTAL then
    trigger_insert(entity_right.name, r.name, e_l, t_l) or
    trigger_update_or_delete(e_l.name, r.name, entity_right, table) or true
  else
   true
  endif;
 }
}
top relation direct_inheritance_to_table {
  e_name : String;
```

```
 t_s : relational::BaseTable;
 checkonly domain eer e_t : eer::Entity {
  name = e_name,
  directInheritanceLinkTarget = direct_inheritance : eer::DirectInheritanceLink{
    source = e_s : eer::Entity{}
  }
 };
 enforce domain rdbm table : relational::BaseTable {
  name = e_name
 };
 when{
     e_t.directInheritanceLinkTarget->size()>0;
     entity_to_table(e_s, t_s) or weak_entity_to_table(e_s, t_s) or
     direct_inheritance_to_table(e_s, t_s) or specialized_inheritance_to_table(e_s, t_s);
 }
 where{
  element_attribute_to_column('', e_t, table);
  element_attribute_compound_to_column('', e_t , table);
  element_attribute_to_pk('', e_t.name, e_t , table);
  element_attribute_compound_to_pk('', e_t.name, e_t , table);
  element_attribute_to_table(e_t.name, e_t.name, e_t, table, e_t);
  element_attribute_compound_to_table(e_t.name, e_t.name, e_t, table, e_t);
  element_attribute_to_pk(e_s.name+'_'+e_t.name+'_', e_t.name, e_s , table);
  element_attribute_compound_to_pk(e_s.name+'_'+e_t.name+'_', e_t.name, e_s , table);
  element_attribute_to_fk(e_s.name+'_'+e_t.name, e_s.name, 'false', table, e_s, t_s);
  element_attribute_compound_to_fk(e_s.name+'_'+e_t.name, e_s.name, 'false', table, e_s, t_s);
  weak_entity_to_pk(e_s.name+'_'+e_t.name, e_t.name, e_s , table);
  weak_entity_to_fk(e_s.name+'_'+e_t.name, e_s.name, 'false', t_s , e_s , table);
  direct_inheritance_to_pk(e_s.name+'_'+e_t.name, e_t.name, e_s, table);
  direct_inheritance_to_fk(e_s.name+'_'+e_t.name,  e_s.name, 'false', t_s ,e_s, table);
  specialized_inheritance_to_pk(e_s.name+'_'+e_t.name, e_t.name, e_s, table);
  specialized_inheritance_to_fk(e_s.name+'_'+e_t.name, e_s.name, 'false',  t_s ,e_s, table);
 }
}

top relation specialized_inheritance_to_table{
 e_name : String;
 t_s : relational::BaseTable;
 checkonly domain eer e_t : eer::Entity {
  name = e_name,
  inheritanceSL = i_SL : eer::InheritanceSL{
   target = inheritance : eer::Inheritance {
    inheritanceGL = inheritance_GL : eer::InheritanceGL{
     source = e_s : eer::Entity{}
    }
   }
  }
 };
 enforce domain rdbm table : relational::BaseTable {
  name = e_name
 };
 when{
  e_t.inheritanceSL->size()>0;
  entity_to_table(e_s, t_s) or weak_entity_to_table(e_s, t_s) or
  direct_inheritance_to_table(e_s, t_s) or specialized_inheritance_to_table(e_s, t_s);
 }
 where{
```

```
   element_attribute_to_column('', e_t, table);
   element_attribute_compound_to_column('', e_t , table);
   element_attribute_to_pk('', e_t.name, e_t , table);
   element_attribute_compound_to_pk('', e_t.name, e_t , table);
   element_attribute_to_table(e_t.name, e_t.name, e_t, table, e_t);
   element_attribute_compound_to_table(e_t.name, e_t.name, e_t, table, e_t);
   element_attribute_to_pk(e_s.name+'_'+e_t.name+'_', e_t.name, e_s , table);
   element_attribute_compound_to_pk(e_s.name+'_'+e_t.name+'_', e_t.name, e_s , table);
   element_attribute_to_fk(e_s.name+'_'+e_t.name, e_s.name, 'false', table, e_s, t_s);
   element_attribute_compound_to_fk(e_s.name+'_'+e_t.name, e_s.name, 'false', table, e_s, t_s);
   weak_entity_to_pk(e_s.name+'_'+e_t.name, e_t.name, e_s , table);
   weak_entity_to_fk(e_s.name+'_'+e_t.name, e_s.name, 'false', t_s , e_s , table);
   direct_inheritance_to_pk(e_s.name+'_'+e_t.name, e_t.name, e_s, table);
   direct_inheritance_to_fk(e_s.name+'_'+e_t.name, e_s.name, 'false', t_s ,e_s, table);
   specialized_inheritance_to_pk(e_s.name+'_'+e_t.name, e_t.name, e_s, table);
   specialized_inheritance_to_fk(e_s.name+'_'+e_t.name, e_s.name, 'false', t_s ,e_s, table);
   if inheritance.disjointness = eer::DisjointnessType::DISJOINT then
    if inheritance_GL.completeness = eer::CompletenessType::TOTAL then
     trigger_inheritance_insert(e_t.name, e_s.name, e_s, t_s) or true
    else
     true
    endif
   else
   if inheritance_GL.completeness = eer::CompletenessType::TOTAL then
    trigger_inheritance_overl_total(e_t.name, e_s.name, e_s, t_s) or true
   else
    true
   endif
   endif;
  }
}

top relation associative_entity_1_1_to_table{
  r_l_l : eer::RelationshipLink;
  r_l_r : eer::RelationshipLink;
  e_l : eer::EntityType;
  e_r : eer::EntityType;
  t_l : relational::BaseTable;
  t_r : relational::BaseTable;
  checkonly domain eer associative_entity : eer::AssociativeEntity {
   contains = r : eer::Relationship {
   }
  };
  enforce domain rdbm t: relational::BaseTable {
   name = a_e.name
  };
  when{
   r.relationshipLink->size()= 2;
   r.relationshipLink->
    select(x : eer::RelationshipLink | x.cardinality = eer::CardinalityType::ONE)->size()=2;
   r_l_l = relationship.relationshipLink->select(x: eer::RelationshipLink |
    (x.attributeToHere = true and x.participation = eer::CompletenessType::TOTAL) or
    (x.attributeToHere = true and x.participation = eer::CompletenessType::PARTIAL) or
    (x.attributeToHere = false and x.participation = eer::CompletenessType::TOTAL))->at(1);
   r_l_r = r.relationshipLink->select(x: eer::RelationshipLink | x <> r_l_l )->at(1);
   r_l_l <> r_l_r;
   e_l = r_l_l.source;
```

```
  e_r = r_l_r.source;
  entity_to_table(e_l, t_l) or weak_entity_to_table(e_l, t_l) or
  direct_inheritance_to_table(e_l, t_l) or specialized_inheritance_to_table(e_l, t_l) or
  associative_entity_1_1_to_table(e_l, t_l) or associative_entity_1_N_to_table(e_l, t_l) or
  associative_entity_N_N_to_table(e_l, t_l);
  entity_to_table(e_r, t_r) or weak_entity_to_table(e_r, t_r) or
  direct_inheritance_to_table(e_r, t_r) or specialized_inheritance_to_table(e_r, t_r) or
  associative_entity_1_1_to_table(e_r, t_r) or associative_entity_1_N_to_table(e_r, t_r) or
  associative_entity_N_N_to_table(e_r, t_r);
}
where{
  element_attribute_to_column('', r, t);
  element_attribute_compound_to_column('', r , t);
  element_attribute_to_table(a_e.name, a_e.name, r, t, a_e);
  element_attribute_compound_to_table(a_e.name, a_e.name, r, t, a_e);
  element_attribute_to_pk(r.name+'_'+e_l.name+'_', a_e.name, e_l, t);
  element_attribute_to_fk(r.name+'_'+e_l.name, e_l.name, 'false', t, e_l, t_l);
  element_attribute_compound_to_pk(r.name+'_'+e_l.name+'_', a_e.name, e_l, t);
  element_attribute_compound_to_fk(r.name+'_'+e_l.name, e_l.name, 'false', t, e_l, t_l);
  direct_inheritance_to_pk(r.name+'_'+e_l.name, associative_entity.name, e_l, t);
  direct_inheritance_to_fk(r.name+'_'+e_l.name, e_l.name, 'false', t_l, e_l, t);
  weak_entity_to_pk(r.name+'_'+e_l.name, a_e.name, e_l, t);
  weak_entity_to_fk(r.name+'_'+e_l.name, e_l.name, 'false', t_l, e_l, t);
  specialized_inheritance_to_pk(r.name+'_'+e_l.name, a_e.name, e_l, t);
  specialized_inheritance_to_fk(r.name+'_'+e_l.name, e_l.name, 'false', t_l, e_l, t);
  associative_entity_1_1_to_pk(r.name+'_'+e_l.name, a_e.name, e_l, t);
  associative_entity_1_1_to_fk(r.name+'_'+e_l.name, e_l.name, 'false', t_l, e_l, t);
  associative_entity_1_N_to_pk(r.name+'_'+e_l.name, a_e.name, e_l, t);
  associative_entity_1_N_to_fk(r.name+'_'+e_l.name, e_l.name, 'false', t_l, e_l, t);
  associative_entity_N_N_to_pk(r.name+'_'+e_l.name, a_e.name, e_l, t);
  associative_entity_N_N_to_fk(r.name+'_'+e_l.name, e_l.name, 'false', t_l, e_l, t);
  element_attribute_to_unique(r.name+'_'+e_r.name+'_', r.name+'_'+e_r.name, e_r, t);
  element_attribute_to_fk(r.name+'_'+e_r.name, e_r.name, 'false', t, e_r, t_r);
  element_attribute_compound_to_unique(r.name+'_'+e_r.name+'_', r.name+'_'+e_r.name, e_r, t);
  element_attribute_compound_to_fk(r.name+'_'+e_r.name, e_r.name, 'false', t, e_r, t_r);
  direct_inheritance_to_unique(r.name+'_'+e_r.name+'_', e_r.name, e_r, t);
  direct_inheritance_to_fk(r.name+'_'+e_r.name, e_r.name, 'false', t_r, e_r, t);
  weak_entity_to_unique(r.name+'_'+e_r.name+'_', r.name+'_'+e_r.name, e_r, t);
  weak_entity_to_fk(r.name+'_'+e_r.name, e_r.name, 'false', t_r, e_r, t);
  specialized_inheritance_to_unique(r.name+'_'+e_r.name+'_', r.name+'_'+e_r.name, e_r, t);
  specialized_inheritance_to_fk(r.name+'_'+e_r.name, e_r.name, 'false', t_r, e_r, table);
  associative_entity_1_1_to_unique(r.name+'_'+e_r.name+'_', r.name+'_'+e_r.name, e_r, t);
  associative_entity_1_1_to_fk(r.name+'_'+e_r.name, e_r.name, 'false', t_r, e_r, t);
  associative_entity_1_N_to_unique(r.name+'_'+e_r.name+'_', r.name+'_'+e_r.name, e_r, t);
  associative_entity_1_N_to_fk(r.name+'_'+e_r.name, e_r.name, 'false', t_r, e_r, t);
  associative_entity_N_N_to_unique(r.name+'_'+e_r.name+'_', r.name+'_'+e_r.name, e_r, t);
  associative_entity_N_N_to_fk(r.name+'_'+e_r.name, e_r.name, 'false', t_r, e_r, t);
  if r_l_l.participation = eer::CompletenessType::TOTAL then
    trigger_insert(associative_entity.name, relationship.name, e_l, t_l) or
    trigger_update_or_delete(e_l.name, relationship.name, associative_entity, table) or true
  else
   true
  endif;
  if r_l_r.participation = eer::CompletenessType::TOTAL then
    trigger_insert(associative_entity.name, relationship.name, e_r, t_r) or
    trigger_update_or_delete(e_r.name, relationship.name, associative_entity, table) or true
  else
```

```
    true
   endif;
   }
}

}
```

Fonte: Autor (2015).

Apêndice B

Este apêndice contém o código EGL utilizado na transformação M2T, do Modelo Relacional para código SQL-Relacional do Capítulo 5.

Quadro 16 - Código EGL para transformação M2T.

```
[% out.print("-- DDL/SQL for PostgreSQL 9.3, released in 25/02/2015. \n\n");
var columnCount : Integer;
var constraintCount : Integer;
var constraintCountTotal : Integer;
var tableCount : Integer;
var tableCountTotal : Integer;
for(element in BaseTable.allInstances()) {
 out.print("CREATE TABLE " + element.name + "( \n");
 columnCount := 1;
 for(column in element.columns){
  out.print("    "+column.name);
  if(column.dataType.name = "STRING" ){
   out.print(" VARCHAR");
   if(column.length <> null and column.length <> 0){
    out.print("("+column.length+")");
   }else{out.print("(255)"); }
  }
  if(column.dataType.name = "INTEGER" ){ out.print(" INTEGER");}
  if(column.dataType.name = "TIMESTAMP" ){ out.print(" TIMESTAMP");}
  if(not(column.nullable)){
   var existPKMember : Boolean;
   existPKMember := false;
   for(constraint in element.constraints){
    if(not(constraint.isKindOf(ForeignKey))){
     if(constraint.isKindOf(PrimaryKey)){
      for(member in constraint.members){
       if(column.name = member.name){ existPKMember := true;}
   }}}}
   if (not(existPKMember)){out.print(" NOT NULL");}
  }
  if(columnCount < element.columns.size()){ out.print(",\n");
  }else{  var existPK : Boolean;
   existPK := false;
   if(element.constraints.size() > 0){
    for(constraint in element.constraints){
     if(constraint.isKindOf(PrimaryKey)){ existPK = true; }
   }}
   if(existPK){ out.print(",\n");
   }else{out.print("\n");}
  }
  columnCount := columnCount + 1;
 }
 //PK and UNIQUE Keys
 constraintCountTotal := element.constraints->select(x | not(x.isKindOf(ForeignKey))).size();
 constraintCount := 1;
 for(constraint in element.constraints){
  if(not(constraint.isKindOf(ForeignKey))){
   if(constraint.isKindOf(PrimaryKey)){ out.print("    PRIMARY KEY (");
   } else {  out.print("    UNIQUE ("); }
   columnCount := 1;
   for(member in constraint.members){
```

```
      if(columnCount < constraint.members.size()){ out.print(member.name+", ");
       }else{  out.print(member.name);}
       columnCount := columnCount+1;
     }
     if(constraintCount < constraintCountTotal){
      if(constraint.isKindOf(PrimaryKey)){ out.print(") ,\n");
      }else{ out.print(") INITIALLY DEFERRED ,\n"); }
     }else{
      if(constraint.isKindOf(PrimaryKey)){ out.print(") \n");
      }else{  out.print(") INITIALLY DEFERRED \n");
     }
     }
     constraintCount := constraintCount + 1;
   }
  }out.print(");\n");
}
// Foreign Keys
for(element in ForeignKey.allInstances()) {
  out.print("ALTER TABLE " + element.baseTable.name + " ADD CONSTRAINT "+element.name+"
FOREIGN KEY (");
  columnCount := 1;
  for(member in element.members){
    if(columnCount < element.members.size()){ out.print(member.name+", ");
    }else{ out.print(member.name);}
    columnCount := columnCount+1;
  }
  out.print(") REFERENCES "+element.referencedTable.name+" (");
  columnCount := 1;
  for(member in element.uniqueConstraint.members){
     if(columnCount < element.uniqueConstraint.members.size()){  out.print(member.name+", ");
     }else{ out.print(member.name);}
     columnCount := columnCount+1;
   }out.print(");\n");
}
for(element in Trigger.allInstances()) {
  if(element.updateType = true and element.insertType = false and element.deleteType = true and
element.actionTime = ActionTimeType#AFTER){
   out.print('\nCREATE FUNCTION FUNCTION_'+element.name+'()\n');
   out.print('RETURNS TRIGGER AS $BODY$\n');
   for(table in element.triggerTables){ out.print('    DECLARE count'+table.name+' INTEGER;\n'); }
   out.print('    BEGIN\n');
   for(table in element.triggerTables){
    out.print('        SELECT COUNT(*) INTO count'+table.name+' \n        FROM
'+element.table.name+'\n');
     out.print('        WHERE ');
     for(constraint in element.table.constraints){
      if(constraint.isKindOf(ForeignKey)){
       if(constraint.referencedTable = table){
         var splitSequence1 : new Sequence;
         var splitSequence2 : new Sequence;
         splitSequence1 := element.name.split("_");
         splitSequence2 := constraint.members.at(0).name.split("_");
         if(splitSequence1.at(1) = splitSequence2.at(0)){
           constraintCountTotal := constraint.uniqueConstraint.members.size();
           constraintCount := 0;
           while (constraintCount < constraintCountTotal){
             out.print(element.table.name+'.'+constraint.members.at(constraintCount).name+' =
OLD.'+constraint.members.at(constraintCount).name);
             if(constraintCount < constraintCountTotal-1){ out.print(' and '); }
             constraintCount := constraintCount + 1;
```

```
            }}}}}
    out.print(';\n\n            IF count'+table.name+' = 0 THEN \n');
    out.print('          RAISE EXCEPTION \''+element.name+'\';\nEND IF;\n\n');
  }
  out.print('       IF (TG_OP = \'DELETE\') THEN \n');
  out.print('          RETURN OLD; \n');
  out.print('       ELSIF (TG_OP = \'UPDATE\') THEN \n');
  out.print('          RETURN NEW;\n');
  out.print('       END IF;\n\n');
  out.print('    END\n');
  out.print('$BODY$\nLANGUAGE plpgsql VOLATILE;\n\n');
  out.print('CREATE CONSTRAINT TRIGGER '+element.name+' ');
  out.print('AFTER DELETE OR UPDATE OF ');
  tableCountTotal := element.triggerTables.size();
  tableCount := 0;
  for(table in element.triggerTables){
   for(constraint in element.table.constraints){
   if(constraint.isKindOf(ForeignKey)){
    if(constraint.referencedTable = table){
     var splitSequence1 : new Sequence;
     var splitSequence2 : new Sequence;
      splitSequence1 := element.name.split("_");
      splitSequence2 := constraint.members.at(0).name.split("_");
      if(splitSequence1.at(1) = splitSequence2.at(0)){
        constraintCountTotal := constraint.uniqueConstraint.members.size();
        constraintCount := 0;
        while (constraintCount < constraintCountTotal){
          out.print(constraint.members.at(constraintCount).name);
          if(constraintCount < (constraintCountTotal-1)){ out.print(', '); }
          constraintCount := constraintCount + 1;
          }}}}}
            if(tableCount < (tableCountTotal-1)){ out.print(', ');  }
          tableCount := tableCount + 1;
  }
  out.print(' ON '+element.table.name+' INITIALLY DEFERRED \n');
  out.print('FOR EACH ROW ');
  out.print('EXECUTE PROCEDURE FUNCTION_'+element.name+'();\n');
  }
%]
```

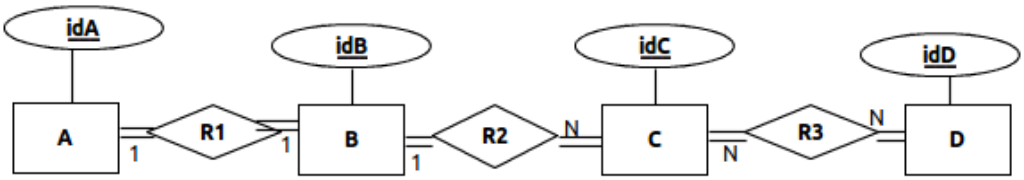
Fonte: Autor (2015).

Apêndice C

Este apêndice contém os esquemas EER (cenários) e seus respectivos códigos SQL-Relacional utilizados no Capítulo 5.

Cenário 3: Restrições de Participação em Relacionamento Binário

Figura 35 - Restrições de Participação em Relacionamento Binário.



Fonte: Autor (2015).

Quadro 17 - Código SQL-Relacional do Cenário 3.

```
-- Criação de Tabelas, Chaves Primárias e demais colunas
CREATE TABLE A(
    idA INTEGER,
    R1_B_idB INTEGER NOT NULL,  -- FK para B, NOT NULL para garantir a
                                -- participação total de A em R1
    PRIMARY KEY (idA) ,
    UNIQUE (R1_B_idB) INITIALLY DEFERRED -- Garantir Cardinalidade 1:1
);
CREATE TABLE B(
    idB INTEGER,
    PRIMARY KEY (idB)
);
CREATE TABLE C(
    idC INTEGER,
    R2_B_idB INTEGER NOT NULL,      -- FK para B
    PRIMARY KEY (idC)
);
CREATE TABLE D(
    idD INTEGER,
    PRIMARY KEY (idD)
);
CREATE TABLE R3(
    R3_C_idC INTEGER,                -- FK para C
    R3_D_idD INTEGER,                -- FK para D
    PRIMARY KEY (R3_C_idC, R3_D_idD)
);
--- Criação das restrições de Chaves Estrangeiras
ALTER TABLE A ADD CONSTRAINT FK_R1_B FOREIGN KEY (R1_B_idB) REFERENCES
B (idB);
ALTER TABLE C ADD CONSTRAINT FK_R2_B FOREIGN KEY (R2_B_idB) REFERENCES
B (idB);
ALTER TABLE R3 ADD CONSTRAINT FK_R3_C FOREIGN KEY (R3_C_idC)
REFERENCES C (idC);
```

```
ALTER TABLE R3 ADD CONSTRAINT FK_R3_D FOREIGN KEY (R3_D_idD)
REFERENCES D (idD);
-- Criação dos Gatilhos para garantir as participações totais
-----------------------------------------------------------------------
/* O gatilho TRG_R1_B garante a participação total de B em R1 depois
de inserir um registro em B. Lança uma exceção quando não houver
registro da FK de B em A */
CREATE FUNCTION FUNCTION_TRG_R1_B()
RETURNS TRIGGER AS $BODY$
    DECLARE countB INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countB
        FROM A
        WHERE A.R1_B_idB = NEW.idB;
        IF countB = 0 THEN
            RAISE EXCEPTION 'TRG_R1_B';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_B AFTER INSERT ON B INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R1_B();
-----------------------------------------------------------------------
/* O gatilho TRG_R1_A garante a participação total de A em R1 depois
de excluir um registro de A ou alterar o valor da FK de B em A. Lança
uma exceção quando não houver registro da FK de B em A */
CREATE FUNCTION FUNCTION_TRG_R1_A()
RETURNS TRIGGER AS $BODY$
    DECLARE countB INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countB
        FROM A
        WHERE A.R1_B_idB = OLD.R1_B_idB;
        IF countB = 0 THEN
            RAISE EXCEPTION 'TRG_R1_A';
        END IF;
        IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_A
AFTER DELETE OR UPDATE OF R1_B_idB ON A INITIALLY DEFERRED FOR EACH
ROW EXECUTE PROCEDURE FUNCTION_TRG_R1_A();
-----------------------------------------------------------------------
/* O gatilho TRG_R2_B garante a participação total de B em R2 depois
de inserir um registro em B. Lança uma exceção quando não houver
registro da FK de B em C */
CREATE FUNCTION FUNCTION_TRG_R2_B()
RETURNS TRIGGER AS $BODY$
    DECLARE countB INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countB
        FROM C
        WHERE C.R2_B_idB = NEW.idB;
        IF countB = 0 THEN
            RAISE EXCEPTION 'TRG_R2_B';
        END IF;
        RETURN NEW;
```

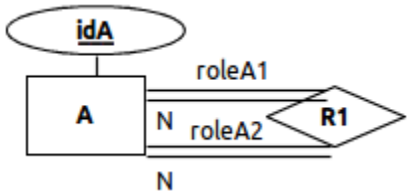
```
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R2_B AFTER INSERT ON B INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R2_B();
-----------------------------------------------------------------------
/* O gatilho TRG_R2_C garante a participação total de C em R2 depois
de excluir um registro de C ou alterar o valor da FK de B em C. Lança
uma exceção quando não houver registro da FK de B em C */
CREATE FUNCTION FUNCTION_TRG_R2_C()
RETURNS TRIGGER AS $BODY$
    DECLARE countB INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countB
        FROM C
        WHERE C.R2_B_idB = OLD.R2_B_idB;
        IF countB = 0 THEN
            RAISE EXCEPTION 'TRG_R2_C';
        END IF;
        IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R2_C
AFTER DELETE OR UPDATE OF R2_B_idB ON C INITIALLY DEFERRED FOR EACH
ROW EXECUTE PROCEDURE FUNCTION_TRG_R2_C();
-----------------------------------------------------------------------
/* O gatilho TRG_R3_D garante a participação total de D em R3 depois
de inserir um registro em D. Lança uma exceção quando não houver
registro da FK de D em R3 */
CREATE FUNCTION FUNCTION_TRG_R3_D()
RETURNS TRIGGER AS $BODY$
    DECLARE countD INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countD
        FROM R3
        WHERE R3.R3_D_idD = NEW.idD;
        IF countD = 0 THEN
            RAISE EXCEPTION 'TRG_R3_D';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R3_D AFTER INSERT ON D INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R3_D();
-----------------------------------------------------------------------
/* O gatilho TRG_R3_C garante a participação total de C em R3 depois
de inserir um registro em C. Lança uma exceção quando não houver
registro da FK de C em R3 */
CREATE FUNCTION FUNCTION_TRG_R3_C()
RETURNS TRIGGER AS $BODY$
    DECLARE countC INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countC
        FROM R3
        WHERE R3.R3_C_idC = NEW.idC;
        IF countC = 0 THEN
            RAISE EXCEPTION 'TRG_R3_C';
        END IF;
```

```
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R3_C AFTER INSERT ON C INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R3_C();
-----------------------------------------------------------------------
/* O gatilho TRG_R3_R3 garante a participação total de C e D em R3
depois de excluir um registro de R3 ou alterar o valor da FK de C ou D
em R3. Lança uma exceção quando não houver registro da FK de C ou da
FK de D em R3 */
CREATE FUNCTION FUNCTION_TRG_R3_R3()
RETURNS TRIGGER AS $BODY$
    DECLARE countD INTEGER;
    DECLARE countC INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countD
        FROM R3
        WHERE R3.R3_D_idD = OLD.R3_D_idD;
        IF countD = 0 THEN
            RAISE EXCEPTION 'TRG_R3_R3';
        END IF;
        SELECT COUNT(*) INTO countC
        FROM R3
        WHERE R3.R3_C_idC = OLD.R3_C_idC;
        IF countC = 0 THEN
            RAISE EXCEPTION 'TRG_R3_R3';
        END IF;
        IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R3_R3 AFTER DELETE OR UPDATE OF
R3_D_idD, R3_C_idC ON R3 INITIALLY DEFERRED FOR EACH ROW EXECUTE
PROCEDURE FUNCTION_TRG_R3_R3();
```

Fonte: Autor (2015).

Cenário 4: Restrições de Participação em Relacionamento Unário

Figura 36 - Restrições de Participação em Relacionamento Unário.



Fonte: Autor (2015).

Quadro 18 - Código SQL-Relacional do Cenário 4.

```
-- Criação de Tabelas, Chaves Primárias e demais colunas
CREATE TABLE A(
    idA INTEGER,
    PRIMARY KEY (idA)
);
```

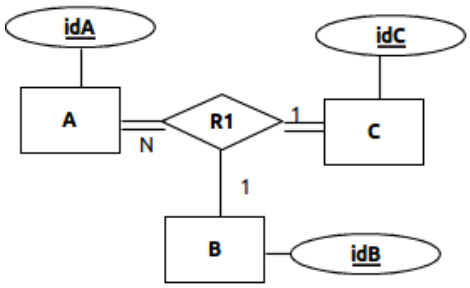
```
CREATE TABLE R1(
    R1_A_roleA1_idA INTEGER,    -- FK para A (via roleA1)
    R1_A_roleA2_idA INTEGER,    -- FK para A (via roleA2)
    PRIMARY KEY (R1_A_roleA1_idA, R1_A_roleA2_idA)
);
--- Criação das restrições de Chaves Estrangeiras
ALTER TABLE R1 ADD CONSTRAINT FK_R1_A_roleA1 FOREIGN KEY
(R1_A_roleA1_idA) REFERENCES A (idA);
ALTER TABLE R1 ADD CONSTRAINT FK_R1_A_roleA2 FOREIGN KEY
(R1_A_roleA2_idA) REFERENCES A (idA);
-- Criação dos Gatilhos para garantir as participações totais
-----------------------------------------------------------------------
/* O gatilho TRG_R1_roleA2_A garante a participação total de A em R1
(via roleA2) depois de inserir um registro em A. Lança uma exceção
quando não houver registro da FK de A (via roleA2) em R1 */
CREATE FUNCTION FUNCTION_TRG_R1_roleA2_A()
RETURNS TRIGGER AS $BODY$
    DECLARE countA INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countA
        FROM R1
        WHERE R1.R1_A_roleA2_idA = NEW.idA;
        IF countA = 0 THEN
            RAISE EXCEPTION 'TRG_R1_roleA2_A';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_roleA2_A AFTER INSERT ON A INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R1_roleA2_A();
-----------------------------------------------------------------------
/* O gatilho TRG_R1_roleA2_R1 garante a participação total de A em R1
(via roleA2) depois de excluir um registro de R1 ou alterar o valor da
FK de A (via roleA2) em R1. Lança uma exceção quando não houver
registro da FK de A (via roleA2) em R1 */
CREATE FUNCTION FUNCTION_TRG_R1_roleA2_R1()
RETURNS TRIGGER AS $BODY$
    DECLARE countA INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countA
        FROM R1
        WHERE R1.R1_A_roleA2_idA = OLD.R1_A_roleA2_idA;
        IF countA = 0 THEN
            RAISE EXCEPTION 'TRG_R1_roleA2_R1';
        END IF;
        IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_roleA2_R1 AFTER DELETE OR UPDATE OF
R1_A_roleA2_idA ON R1 INITIALLY DEFERRED
FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R1_roleA2_R1();
-----------------------------------------------------------------------
/* O gatilho TRG_R1_roleA1_A garante a participação total de A em R1
(via roleA1) depois de inserir um registro em A. Lança uma exceção
quando não houver registro da FK de A (via roleA1) em R1 */
CREATE FUNCTION FUNCTION_TRG_R1_roleA1_A()
RETURNS TRIGGER AS $BODY$
```

```
    DECLARE countA INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countA
        FROM R1
        WHERE R1.R1_A_roleA1_idA = NEW.idA;
        IF countA = 0 THEN
            RAISE EXCEPTION 'TRG_R1_roleA1_A';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_roleA1_A AFTER INSERT ON A INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R1_roleA1_A();
-------------------------------------------------------------------------
/* O gatilho TRG_R1_roleA1_R1 garante a participação total de A em R1
(via roleA1) depois de excluir um registro de R1 ou alterar o valor da
FK de A (via roleA1) em R1. Lança uma exceção quando não houver
registro da FK de A (via roleA1) em R1 */
CREATE FUNCTION FUNCTION_TRG_R1_roleA1_R1()
RETURNS TRIGGER AS $BODY$
    DECLARE countA INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countA
        FROM R1
        WHERE R1.R1_A_roleA1_idA = OLD.R1_A_roleA1_idA;
        IF countA = 0 THEN
            RAISE EXCEPTION 'TRG_R1_roleA1_R1';
        END IF;
        IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_roleA1_R1 AFTER DELETE OR UPDATE OF
R1_A_roleA1_idA ON R1 INITIALLY DEFERRED
FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R1_roleA1_R1();
```

Fonte: Autor (2015).

Cenário 5: Restrições de Participação em Relacionamento N-ário

Figura 37 - Restrições de Participação em Relacionamento N-ário.



Fonte: Autor (2015).

Quadro 19 - Código SQL-Relacional do Cenário 5.

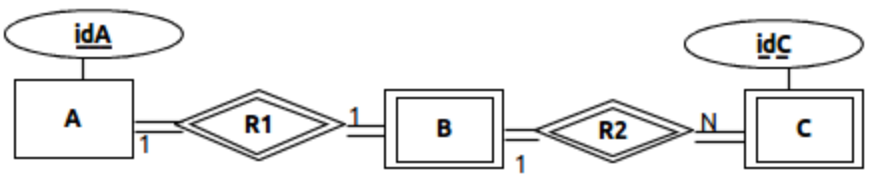
```
-- Criação de Tabelas, Chaves Primárias e demais colunas
CREATE TABLE A(
    idA INTEGER,
    PRIMARY KEY (idA)
);
CREATE TABLE R1(
    R1_A_idA INTEGER,           -- FK para A
    R1_B_idB INTEGER,           -- FK para B
    R1_C_idC INTEGER NOT NULL, -- FK para C (chosenLink = true)
    UNIQUE (R1_A_idA, R1_C_idC) INITIALLY DEFERRED,
    -- Para garantir a cardinalidade 1:1:N
    PRIMARY KEY (R1_A_idA, R1_B_idB)
);
CREATE TABLE B(
    idB INTEGER,
    PRIMARY KEY (idB)
);
CREATE TABLE C(
    idC INTEGER,
    PRIMARY KEY (idC)
);
--- Criação das restrições de Chaves Estrangeiras
ALTER TABLE R1 ADD CONSTRAINT FK_R1_A FOREIGN KEY (R1_A_idA)
REFERENCES A (idA);
ALTER TABLE R1 ADD CONSTRAINT FK_R1_B FOREIGN KEY (R1_B_idB)
REFERENCES B (idB);
ALTER TABLE R1 ADD CONSTRAINT FK_R1_C FOREIGN KEY (R1_C_idC)
REFERENCES C (idC);
Criação dos Gatilhos para garantir as participações totais
-----------------------------------------------------------------------
/* O gatilho TRG_R1_A garante a participação total de A em R1 depois
de inserir um registro em A. Lança uma exceção quando não houver
registro da FK de A em R1 */
CREATE FUNCTION FUNCTION_TRG_R1_A()
RETURNS TRIGGER AS $BODY$
    DECLARE countA INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countA
        FROM R1
        WHERE R1.R1_A_idA = NEW.idA;
        IF countA = 0 THEN
            RAISE EXCEPTION 'TRG_R1_A';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_A AFTER INSERT ON A INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R1_A();
-----------------------------------------------------------------------
/* O gatilho TRG_R1_C garante a participação total de C em R1 depois
de inserir um registro em C. Lança uma exceção quando não houver
registro da FK de C em R1 */
CREATE FUNCTION FUNCTION_TRG_R1_C()
RETURNS TRIGGER AS $BODY$
    DECLARE countC INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countC
        FROM R1
        WHERE R1.R1_C_idC = NEW.idC;
```

```
        IF countC = 0 THEN
            RAISE EXCEPTION 'TRG_R1_C';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_C AFTER INSERT ON C INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R1_C();
-----------------------------------------------------------------------
/* O gatilho TRG_R1_R1 garante a participação total de A e C em R1
depois de excluir um registro de R1 ou alterar o valor da FK de A ou C
em R1. Lança uma exceção quando não houver registro da FK de A ou da
FK de C em R1 */
CREATE FUNCTION FUNCTION_TRG_R1_R1()
RETURNS TRIGGER AS $BODY$
    DECLARE countA INTEGER;
    DECLARE countC INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countA
        FROM R1
        WHERE R1.R1_A_idA = OLD.R1_A_idA;
        IF countA = 0 THEN
            RAISE EXCEPTION 'TRG_R1_R1';
        END IF;
        SELECT COUNT(*) INTO countC
        FROM R1
        WHERE R1.R1_C_idC = OLD.R1_C_idC;
        IF countC = 0 THEN
            RAISE EXCEPTION 'TRG_R1_R1';
        END IF;
        IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_R1 AFTER DELETE OR UPDATE OF
R1_A_idA, R1_C_idC ON R1 INITIALLY DEFERRED FOR EACH ROW EXECUTE
PROCEDURE FUNCTION_TRG_R1_R1();
```

Fonte: Autor (2015).

Cenário 6: Restrições de Participação em Relacionamento Identificador

Figura 38 - Restrições de Participação em Relacionamento Identificador.



Fonte: Autor (2015).

Quadro 20 - Código SQL-Relacional do Cenário 6.

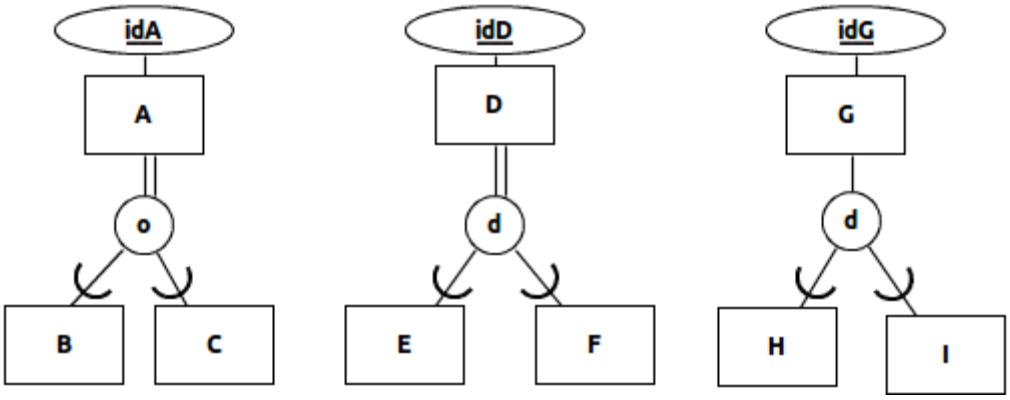
```
-- Criação de Tabelas, Chaves Primárias e demais colunas
CREATE TABLE A(
    idA INTEGER,
    PRIMARY KEY (idA)
);
CREATE TABLE B(
    R1_A_idA INTEGER,         -- FK para A
    PRIMARY KEY (R1_A_idA)
);
CREATE TABLE C(
    idC INTEGER,
    R2_B_A_idA INTEGER,        -- FK para B
    PRIMARY KEY (idC, R2_B_A_idA)
);
-- Criação das restrições de Chaves Estrangeiras
ALTER TABLE B ADD CONSTRAINT FK_R1_A FOREIGN KEY (R1_A_idA) REFERENCES
A (idA);
ALTER TABLE C ADD CONSTRAINT FK_R2_B FOREIGN KEY (R2_B_A_idA)
REFERENCES B (R1_A_idA);
-- Criação dos Gatilhos para garantir as participações totais
-----------------------------------------------------------------------
/* O gatilho TRG_R2_B garante a participação total de B em R2 depois
de inserir um registro em B. Lança uma exceção quando não houver
registro da FK de B em C */
CREATE FUNCTION FUNCTION_TRG_R2_B()
RETURNS TRIGGER AS $BODY$
    DECLARE countB INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countB
        FROM C
        WHERE C.R2_B_A_idA = NEW.R1_A_idA;
        IF countB = 0 THEN
            RAISE EXCEPTION 'TRG_R2_B';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R2_B AFTER INSERT ON B INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R2_B();
-----------------------------------------------------------------------
/* O gatilho TRG_R2_C garante a participação total de B em R2 depois
de excluir um registro de C ou alterar o valor da FK de B em C. Lança
uma exceção quando não houver registro da FK de B em C */
CREATE FUNCTION FUNCTION_TRG_R2_C()
RETURNS TRIGGER AS $BODY$
    DECLARE countB INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countB
        FROM C
        WHERE C.R2_B_A_idA = OLD.R2_B_A_idA;
        IF countB = 0 THEN
            RAISE EXCEPTION 'TRG_R2_C';
        END IF;
        IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
```

```
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R2_C AFTER DELETE OR UPDATE OF
R2_B_A_idA ON C INITIALLY DEFERRED FOR EACH ROW EXECUTE PROCEDURE
FUNCTION_TRG_R2_C();
--------------------------------------------------------------------
/* O gatilho TRG_R1_A garante a participação total de A em R1 depois
de inserir um registro em A. Lança uma exceção quando não houver
registro da FK de A em B */
CREATE FUNCTION FUNCTION_TRG_R1_A()
RETURNS TRIGGER AS $BODY$
    DECLARE countA INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countA
        FROM B
        WHERE B.R1_A_idA = NEW.idA;
        IF countA = 0 THEN
            RAISE EXCEPTION 'TRG_R1_A';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_A AFTER INSERT ON A INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R1_A();
--------------------------------------------------------------------
/* O gatilho TRG_R1_B garante a participação total de A em R1 depois
de excluir um registro de B ou alterar o valor da FK de A em B. Lança
uma exceção quando não houver registro da FK de A em B */
CREATE FUNCTION FUNCTION_TRG_R1_B()
RETURNS TRIGGER AS $BODY$
    DECLARE countA INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countA
        FROM B
        WHERE B.R1_A_idA = OLD.R1_A_idA;
        IF countA = 0 THEN
            RAISE EXCEPTION 'TRG_R1_B';
        END IF;
        IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_B AFTER DELETE OR UPDATE OF R1_A_idA
ON B INITIALLY DEFERRED FOR EACH ROW EXECUTE PROCEDURE
FUNCTION_TRG_R1_B();
```

Fonte: Autor (2015).

Cenário 7: Restrições de Completude e Disjunção em Herança

Figura 39 - Restrições de Completude e Disjunção em Herança.



Fonte: Autor (2015).

Quadro 21 - Código SQL-Relacional do Cenário 7.

```
-- Criação de Tabelas, Chaves Primárias e demais colunas
CREATE TABLE A(
    idA INTEGER,
    PRIMARY KEY (idA)
);
CREATE TABLE C(
    A_C_idA INTEGER,       -- FK para A
    PRIMARY KEY (A_C_idA)
);
CREATE TABLE B(
    A_B_idA INTEGER,       -- FK para A
    PRIMARY KEY (A_B_idA)
);
CREATE TABLE D(
    idD INTEGER,
    PRIMARY KEY (idD)
);
CREATE TABLE F(
    D_F_idD INTEGER,       -- FK para D
    PRIMARY KEY (D_F_idD)
);
CREATE TABLE E(
    D_E_idD INTEGER,       -- FK para D
    PRIMARY KEY (D_E_idD)
);
CREATE TABLE G(
    idG INTEGER,
    PRIMARY KEY (idG)
);
CREATE TABLE I(
    G_I_idG INTEGER,        -- FK para G
    PRIMARY KEY (G_I_idG)
);
CREATE TABLE H(
    G_H_idG INTEGER,        -- FK para G
    PRIMARY KEY (G_H_idG)
);
```

```
-- Criação das restrições de Chaves Estrangeiras
ALTER TABLE C ADD CONSTRAINT FK_A_C FOREIGN KEY (A_C_idA) REFERENCES A
(idA);
ALTER TABLE B ADD CONSTRAINT FK_A_B FOREIGN KEY (A_B_idA) REFERENCES A
(idA);
ALTER TABLE F ADD CONSTRAINT FK_D_F FOREIGN KEY (D_F_idD) REFERENCES D
(idD);
ALTER TABLE E ADD CONSTRAINT FK_D_E FOREIGN KEY (D_E_idD) REFERENCES D
(idD);
ALTER TABLE I ADD CONSTRAINT FK_G_I FOREIGN KEY (G_I_idG) REFERENCES G
(idG);
ALTER TABLE H ADD CONSTRAINT FK_G_H FOREIGN KEY (G_H_idG) REFERENCES G
(idG);
-- Criação dos Gatilhos para garantir as participações totais
-----------------------------------------------------------------------
/* O gatilho TRG_A_A garante a participação total de A na herança
depois de inserir um registro em A. Lança uma exceção quando não
houver registro da FK de A em C ou B */
CREATE FUNCTION FUNCTION_TRG_A_A()
RETURNS TRIGGER AS $BODY$
    DECLARE countC INTEGER;
    DECLARE countB INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countC
        FROM C
        WHERE C.A_C_idA = NEW.idA;
        SELECT COUNT(*) INTO countB
        FROM B
        WHERE B.A_B_idA = NEW.idA;
        IF countC + countB = 0 THEN
            RAISE EXCEPTION 'TRG_A_A';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_A_A AFTER INSERT ON A INITIALLY DEFERRED
FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_A_A();
-----------------------------------------------------------------------
/* O gatilho TRG_A_B garante a participação total de A na herança
depois de excluir um registro de B ou alterar o valor da FK de A em B.
Lança uma exceção quando não houver registro da FK de A em C */
CREATE FUNCTION FUNCTION_TRG_A_B()
RETURNS TRIGGER AS $BODY$
    DECLARE countC INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countC
        FROM C
        WHERE C.A_C_idA = OLD.A_B_idA;
        IF countC = 0 THEN
            RAISE EXCEPTION 'TRG_A_B';
        END IF;
        IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_A_B AFTER DELETE OR UPDATE OF A_B_idA ON
B INITIALLY DEFERRED FOR EACH ROW EXECUTE PROCEDURE
FUNCTION_TRG_A_B();
```

```
-------------------------------------------------------------------------
/* O gatilho TRG_A_C garante a participação total de A na herança
depois de excluir um registro de C ou alterar o valor da FK de A em C.
Lança uma exceção quando não houver registro da FK de A em B */
CREATE FUNCTION FUNCTION_TRG_A_C()
RETURNS TRIGGER AS $BODY$
    DECLARE countB INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countB
        FROM B
        WHERE B.A_B_idA = OLD.A_C_idA;
        IF countB = 0 THEN
            RAISE EXCEPTION 'TRG_A_C';
        END IF;
        IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_A_C AFTER DELETE OR UPDATE OF A_C_idA ON
C INITIALLY DEFERRED FOR EACH ROW EXECUTE PROCEDURE
FUNCTION_TRG_A_C();
-------------------------------------------------------------------------
/* O gatilho TRG_G_H garante disjunção disjunta de G na herança depois
de inserir um registro em F ou alterar o valor da FK de G em H. Lança
uma exceção quando ao inserir ou  atualizar o novo registro de H
houver registro da FK de G em I */
CREATE FUNCTION FUNCTION_TRG_G_H()
RETURNS TRIGGER AS $BODY$
    DECLARE countI INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countI
        FROM I
        WHERE I.G_I_idG = NEW.G_H_idG;
        IF countI <> 0 THEN
            RAISE EXCEPTION 'TRG_G_H';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_G_H   AFTER INSERT OR UPDATE OF G_H_idG
ON H INITIALLY DEFERRED FOR EACH ROW EXECUTE PROCEDURE
FUNCTION_TRG_G_H();
-------------------------------------------------------------------------
/* O gatilho TRG_G_I garante disjunção disjunta de G na herança depois
de inserir um registro em I ou alterar o valor da FK de G em I. Lança
uma exceção quando ao inserir ou  atualizar o novo registro de I
houver registro da FK de G em F */
CREATE FUNCTION FUNCTION_TRG_G_I()
RETURNS TRIGGER AS $BODY$
    DECLARE countH INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countH
        FROM H
        WHERE H.G_H_idG = NEW.G_I_idG;
        IF countH <> 0 THEN
            RAISE EXCEPTION 'TRG_G_I';
        END IF;
        RETURN NEW;
```

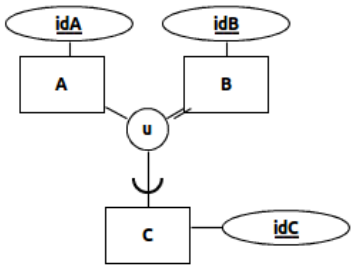
```
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_G_I AFTER INSERT OR UPDATE OF G_I_idG ON
I INITIALLY DEFERRED FOR EACH ROW EXECUTE PROCEDURE
FUNCTION_TRG_G_I();
-----------------------------------------------------------------------
/* O gatilho TRG_D_D garante a participação total de D na herança
depois de inserir um registro em D. Lança uma exceção quando não
houver registro da FK de D em F ou E */
CREATE FUNCTION FUNCTION_TRG_D_D()
RETURNS TRIGGER AS $BODY$
    DECLARE countF INTEGER;
    DECLARE countE INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countF
        FROM F
        WHERE F.D_F_idD = NEW.idD;
        SELECT COUNT(*) INTO countE
        FROM E
        WHERE E.D_E_idD = NEW.idD;
        IF countF + countE = 0 THEN
            RAISE EXCEPTION 'TRG_D_D';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT  TRIGGER TRG_D_D AFTER INSERT ON D INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_D_D();
-----------------------------------------------------------------------
/* O gatilho TRG_D_E garante a participação total e a disjunção
disjunta de D na herança depois de excluir um registro de E, alterar o
valor da FK de D em E ou inserir um registro em E. Lança uma exceção
quando ao deletar ou atualizar o antigo registro de E não houver
registro da FK de D em F ou lança uma exceção quando ao inserir ou
atualizar o novo registro de E houver registro da FK de D em F */
CREATE FUNCTION FUNCTION_TRG_D_E()
RETURNS TRIGGER AS $BODY$
    DECLARE countF INTEGER;
    BEGIN
        IF (TG_OP = 'DELETE' OR TG_OP = 'UPDATE') THEN
            SELECT COUNT(*) INTO countF
            FROM F
            WHERE F.D_F_idD = OLD.D_E_idD;
            IF countF = 0 THEN
                RAISE EXCEPTION 'TRG_D_E';
            END IF;
        END IF;
        IF (TG_OP = 'INSERT' OR TG_OP = 'UPDATE') THEN
            SELECT COUNT(*) INTO countF
            FROM F
            WHERE F.D_F_idD = NEW.D_E_idD;
            IF countF <> 0 THEN
                RAISE EXCEPTION 'TRG_D_E';
            END IF;
        END IF;
       IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE' OR TG_OP = 'INSERT') THEN
            RETURN NEW;
        END IF;
    END
```

```
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_D_E AFTER  INSERT OR DELETE OR UPDATE OF
D_E_idD ON E INITIALLY DEFERRED FOR EACH ROW EXECUTE PROCEDURE
FUNCTION_TRG_D_E();
--------------------------------------------------------------------------
/* O gatilho TRG_D_F garante a participação total e a disjunção
disjunta de D na herança depois de excluir um registro de F, alterar o
valor da FK de D em F ou inserir um registro em F. Lança uma exceção
quando ao deletar ou atualizar o antigo registro de F não houver
registro da FK de D em E ou lança uma exceção quando ao inserir ou
atualizar o novo registro de F houver registro da FK de D em E */
CREATE FUNCTION FUNCTION_TRG_D_F()
RETURNS TRIGGER AS $BODY$
    DECLARE countE INTEGER;
    BEGIN
        IF (TG_OP = 'DELETE' OR TG_OP = 'UPDATE') THEN
            SELECT COUNT(*) INTO countE
            FROM E
            WHERE E.D_E_idD = OLD.D_F_idD;
            IF countE = 0 THEN
                RAISE EXCEPTION 'TRG_D_F';
            END IF;
        END IF;
        IF (TG_OP = 'INSERT' OR TG_OP = 'UPDATE') THEN
            SELECT COUNT(*) INTO countE
            FROM E
            WHERE E.D_E_idD = NEW.D_F_idD;
            IF countE <> 0 THEN
                RAISE EXCEPTION 'TRG_D_F';
            END IF;
        END IF;
       IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE' OR TG_OP = 'INSERT') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_D_F AFTER  INSERT OR DELETE OR UPDATE OF
D_F_idD ON F INITIALLY DEFERRED FOR EACH ROW EXECUTE PROCEDURE
FUNCTION_TRG_D_F();
```

Fonte: Autor (2015).

Cenário 8: Restrições de Completude em Categoria

Figura 40 - Restrições de Completude em Categoria.



Fonte: Autor (2015).

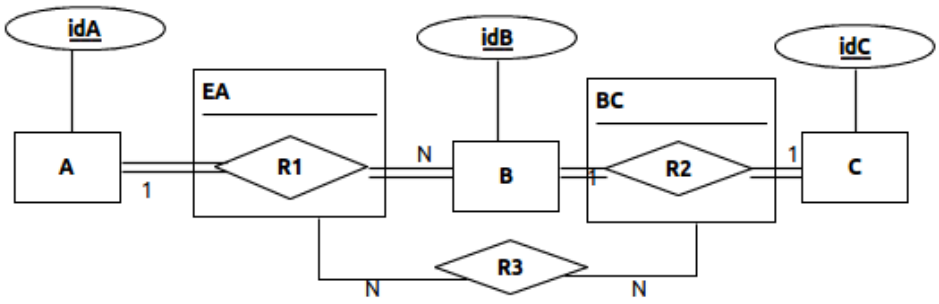
Quadro 22 - Código SQL-Relacional do Cenário 8.

```
-- Criação de Tabelas, Chaves Primárias e demais colunas
CREATE TABLE A(
    idA INTEGER,
    C_A_idC VARCHAR(255),  -- FK para C, NULL para Completude parcial
    PRIMARY KEY (idA)
);
CREATE TABLE B(
    idB INTEGER,
    C_B_idC VARCHAR(255) NOT NULL,  -- FK para C, NOT NULL para
Completude total
    PRIMARY KEY (idB)
);
CREATE TABLE C(
    idC VARCHAR(255),
    PRIMARY KEY (idC)
);

--- Criação das restrições de Chaves Estrangeiras
ALTER TABLE B ADD CONSTRAINT FK_C_B FOREIGN KEY (C_B_idC) REFERENCES C
(idC);
ALTER TABLE A ADD CONSTRAINT FK_C_A FOREIGN KEY (C_A_idC) REFERENCES C
(idC);
```

Fonte: Autor (2015).

Cenário 9: Restrições de Participação em Entidade Associativa

Figura 41 - Restrições de Participação em Entidade Associativa.



Fonte: Autor (2015).

Quadro 23 - Código SQL-Relacional do Cenário 9.

```
-- Criação de Tabelas, Chaves Primárias e demais colunas
CREATE TABLE A(
    idA INTEGER,
    PRIMARY KEY (idA)
);
CREATE TABLE B(
    idB INTEGER,
    PRIMARY KEY (idB)
);
CREATE TABLE EA(
    R1_A_idA INTEGER NOT NULL,  -- FK para A
    R1_B_idB INTEGER,           -- FK para B
    PRIMARY KEY (R1_B_idB)
```

```
);
CREATE TABLE C(
    idC INTEGER,
    PRIMARY KEY (idC)
);
CREATE TABLE BC(
    R2_C_idC INTEGER,              -- FK para C
    R2_B_idB INTEGER NOT NULL,     -- FK para B
    PRIMARY KEY (R2_C_idC),
    UNIQUE (R2_B_idB) INITIALLY DEFERRED -- Cardinalidade 1:1
);
CREATE TABLE R3(
    R3_EA_idB INTEGER,
    R3_BC_idC INTEGER,
    PRIMARY KEY (R3_EA_idB, R3_BC_idC)
);
--- Criação das restrições de Chaves Estrangeiras
ALTER TABLE EA ADD CONSTRAINT FK_R1_A FOREIGN KEY (R1_A_idA)
REFERENCES A (idA);
ALTER TABLE EA ADD CONSTRAINT FK_R1_B FOREIGN KEY (R1_B_idB)
REFERENCES B (idB);
ALTER TABLE BC ADD CONSTRAINT FK_R2_C FOREIGN KEY (R2_C_idC)
REFERENCES C (idC);
ALTER TABLE BC ADD CONSTRAINT FK_R2_B FOREIGN KEY (R2_B_idB)
REFERENCES B (idB);
ALTER TABLE R3 ADD CONSTRAINT FK_R3_BC FOREIGN KEY (R3_BC_idC)
REFERENCES BC (R2_C_idC);
ALTER TABLE R3 ADD CONSTRAINT FK_R3_EA FOREIGN KEY (R3_EA_idB)
REFERENCES EA (R1_B_idB);
--- Criação dos Gatilhos para garantir as participações totais
-----------------------------------------------------------------------
/* O gatilho TRG_R1_B garante a participação total de B em R1 depois
de inserir um registro em B. Lança uma exceção quando não houver
registro da FK de B em EA */
CREATE FUNCTION FUNCTION_TRG_R1_B()
RETURNS TRIGGER AS $BODY$
    DECLARE countB INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countB
        FROM EA
        WHERE EA.R1_B_idB = NEW.idB;
        IF countB = 0 THEN
            RAISE EXCEPTION 'TRG_R1_B';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_B AFTER INSERT ON B INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R1_B();
-----------------------------------------------------------------------
/* O gatilho TRG_R1_A garante a participação total de A em R1 depois
de inserir um registro em A. Lança uma exceção quando não houver
registro da FK de A em EA */
CREATE FUNCTION FUNCTION_TRG_R1_A()
RETURNS TRIGGER AS $BODY$
    DECLARE countA INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countA
        FROM EA
        WHERE EA.R1_A_idA = NEW.idA;
        IF countA = 0 THEN
```

```
            RAISE EXCEPTION 'TRG_R1_A';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_A AFTER INSERT ON A INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R1_A();
-----------------------------------------------------------------------
/* O gatilho TRG_R1_EA garante a participação total de A e B em R1
depois de excluir um registro de EA ou alterar o valor da FK de A ou B
em EA. Lança uma exceção quando não houver registro da FK de A ou da
FK de B em EA */
CREATE FUNCTION FUNCTION_TRG_R1_EA()
RETURNS TRIGGER AS $BODY$
    DECLARE countB INTEGER;
    DECLARE countA INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countB
        FROM EA
        WHERE EA.R1_B_idB = OLD.R1_B_idB;
        IF countB = 0 THEN
            RAISE EXCEPTION 'TRG_R1_EA';
        END IF;
        SELECT COUNT(*) INTO countA
        FROM EA
        WHERE EA.R1_A_idA = OLD.R1_A_idA;
        IF countA = 0 THEN
            RAISE EXCEPTION 'TRG_R1_EA';
        END IF;
        IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R1_EA AFTER DELETE OR UPDATE OF
R1_B_idB, R1_A_idA ON EA INITIALLY DEFERRED FOR EACH ROW EXECUTE
PROCEDURE FUNCTION_TRG_R1_EA();
-----------------------------------------------------------------------
/* O gatilho TRG_R2_B garante a participação total de B em R2 depois
de inserir um registro em B. Lança uma exceção quando não houver
registro da FK de B em BC */
CREATE FUNCTION FUNCTION_TRG_R2_B()
RETURNS TRIGGER AS $BODY$
    DECLARE countB INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countB
        FROM BC
        WHERE BC.R2_B_idB = NEW.idB;
        IF countB = 0 THEN
            RAISE EXCEPTION 'TRG_R2_B';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R2_B AFTER INSERT ON B INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R2_B();
-----------------------------------------------------------------------
/* O gatilho TRG_R2_C garante a participação total de C em R2 depois
de inserir um registro em C. Lança uma exceção quando não houver
```

```
registro da FK de C em BC */
CREATE FUNCTION FUNCTION_TRG_R2_C()
RETURNS TRIGGER AS $BODY$
    DECLARE countC INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countC
        FROM BC
        WHERE BC.R2_C_idC = NEW.idC;
        IF countC = 0 THEN
            RAISE EXCEPTION 'TRG_R2_C';
        END IF;
        RETURN NEW;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R2_C AFTER INSERT ON C INITIALLY
DEFERRED FOR EACH ROW EXECUTE PROCEDURE FUNCTION_TRG_R2_C();
-----------------------------------------------------------------------
/* O gatilho TRG_R2_BC garante a participação total de C e B em R2
depois de excluir um registro de BC ou alterar o valor da FK de C ou B
em BC. Lança uma exceção quando não houver registro da FK de C ou da
FK de B em BC */
CREATE FUNCTION FUNCTION_TRG_R2_BC()
RETURNS TRIGGER AS $BODY$
    DECLARE countB INTEGER;
    DECLARE countC INTEGER;
    BEGIN
        SELECT COUNT(*) INTO countB
        FROM BC
        WHERE BC.R2_B_idB = OLD.R2_B_idB;
        IF countB = 0 THEN
            RAISE EXCEPTION 'TRG_R2_BC';
        END IF;
        SELECT COUNT(*) INTO countC
        FROM BC
        WHERE BC.R2_C_idC = OLD.R2_C_idC;
        IF countC = 0 THEN
            RAISE EXCEPTION 'TRG_R2_BC';
        END IF;
        IF (TG_OP = 'DELETE') THEN
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            RETURN NEW;
        END IF;
    END
$BODY$ LANGUAGE plpgsql VOLATILE;
CREATE CONSTRAINT TRIGGER TRG_R2_BC AFTER DELETE OR UPDATE OF
R2_B_idB, R2_C_idC ON BC INITIALLY DEFERRED FOR EACH ROW EXECUTE
PROCEDURE FUNCTION_TRG_R2_BC();
```

Fonte: Autor (2015).