FERNANDO CESAR KLOSS

MOTOR DE TRANSFORMAÇÕES BASEADO EM *MAPREDUCE*

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Marcos Didonet Del Fabro

CURITIBA

2013

K66m Kloss, Fernando Cesar
Motor de transformações baseado em *MapReduce* / Fernando Cesar Kloss. – Curitiba, 2013.
59f. : il. color. ; 30 cm.

Dissertação (mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática, 2013.

Orientador: Marcos Didonet Del Fabro.
Bibliografia: p. 53-59.

1. Modelagem de processos. 2. Software -- Desenvolvimento. I. Universidade Federal do Paraná. II. Fabro, Marcos Didonet del. III. Título.

CDD: 003.71

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Fernando Cesar Kloss, avaliamos o trabalho intitulado, *"Motor de transformações baseado em MapReduce"*, cuja defesa foi realizada no dia 20 de dezembro de 2013, às 14:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela:

(X) **aprovação** do candidato. ( ) **reprovação** do candidato.

Curitiba, 20 de dezembro de 2013.

Prof. Dr. Marcos Didonet Del Fabro
DINF/UFPR – Orientador

Profa. Dra. Nadia Puchalski Kozievitch
UTFPR – Membro Externo

Profa. Dra. Leticia Mara Peres
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

FERNANDO CESAR KLOSS

MOTOR DE TRANSFORMAÇÕES BASEADO EM *MAPREDUCE*

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre no Programa de Pós-Graduação em Informática da Universidade Federal do Paraná, pela Comissão formada pelos professores:

Orientador: Prof. Dr. Marcos Didonet Del Fabro
Departamento de Informática, UFPR

Prof[a]. Dra. Nadia Puchalski Kozievitch
UTFPR - membro externo

Prof[a]. Dra. Leticia Mara Peres
UFPR - membro interno

Curitiba, 20 de dezembro de 2013

FERNANDO CESAR KLOSS

MOTOR DE TRANSFORMAÇÕES BASEADO EM *MAPREDUCE*

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Marcos Didonet Del Fabro

CURITIBA

2013

Dedico está dissertação a minha mãe Iolanda, por todo seu apoio, dedicação e incentivo em minha educação. E a minha futura esposa Carla pela compreensão e paciência durante todo o tempo que estive dedicado a este trabalho.

AGRADECIMENTOS

Ao professor e orientador Marcos Didonet Del Fabro, o agradecimento pelos direcionamentos e por sua atenção e ensejo de aprimoramento deste trabalho.

Agradeço aos amigos Paulo Capeller e Edson Ramiro pelo auxílio na configuração do ambiente de testes.

A Universidade Federal do Paraná e ao Departamento de Informática, professores e funcionários.

Aos colegas do Instituto Federal do Paraná que me apoiaram durante a realização deste trabalho.

Agradeço a Deus por mais uma conquista em minha vida.

Sumário

Lista de Figuras

Lista de Tabelas

LISTA DE LISTAGENS

LISTA DE SIGLAS

API - *Application Programming Interface*

ATL - *Atlas Transformation Language*

COD - *Connected Data Objects*

EMF - *Eclipse Modeling Framework*

ER - *Entity Relationship*

ETL - *Epsilon Transformation Language*

GME - *Generic Modeling Environment*

GReAT - *Graph Rewriting and Transformation*

HDFS - *Hadoop Distributed File System*

M2M - *Model-to-Model*

MDA - *Model Driven Architecture*

MDE - *Model-Driven Engineering*

OCL - *Object Constraint Language*

OMG - *Objetc Management Group*

ORM - *Object-Relational Mapping*

QVT - *Query/View/Transformation*

UML - *Unified Modeling Language*

URI - *Uniform Resource Identifier*

VIATRA - *VIsual Automated model TRAnsformations*

VM - *Virtual Machine*

XMI - *XML Metadata Interchange*

XML - *eXtensible Markup Language*

RESUMO

A busca por agilidade no processo de desenvolvimento de software tem impulsionado a crescente adoção de tecnologias, paradigmas e abordagens baseada em modelos (*Model-Driven Engineering*). Essas soluções mudam o foco de codificação para modelagem, onde modelos são utilizados para descrever diferentes aspectos de um sistema em diferentes níveis de abstração. Uma série de linguagens, padrões e ferramentas surgiram para automatizar a construção e modificação de modelos e assim apoiar a principal operação executada neste cenário que são as transformações de modelos. A inserção de grandes modelos neste contexto evidenciou uma limitação dessa metodologia, a capacidade de tratar modelos com esta característica. Problemas de escalabilidade surgem quando modelos da ordem de milhares de elementos são utilizados em processos de desenvolvimento de software. Trabalhos recentes, visando desenvolver soluções para o problema de escalabilidade, tem explorado e focado em diferentes abordagens como armazenamento, fragmentação e persistência de modelos, porém pouco se tem visto em relação a ferramentas de transformação de modelos. Com base em trabalhos feitos em outros domínios, desenvolvemos um mecanismo de transformação de modelos executando de forma distribuída em uma nuvem. A solução consiste na adaptação de uma ferramenta de transformação de modelos para execução distribuída, através da integração com MapReduce. Duas implementações distintas arquiteturalmente são apresentadas, uma baseada em regras de transformação e outra baseada em operações de transformação de modelos. Os resultados obtidos são promissores especialmente para transformação de modelos grandes e complexos.

ABSTRACT

The search for agility in software development process has driven the increasing adoption of technologies, paradigms and approaches based on models (Model-Driven Engineering). These solutions change the focus from coding to modeling where models are used to describe different aspects of a system on different levels of abstraction. A variety of languages, standards and tools have emerged to automate the construction and modification of models where the main operation performed in this scenario are the model transformations. The insertion of large models in this context showed a limitation of this methodology, the ability to handle large models. Scalability problems arise when models on the order of thousands of elements are used in software development processes. Recent works in order to develop solutions to the problem of scalability has focused and explored different approaches as storage, fragmentation and persistence models, but for model transformation tools have seen very little action. Based on works done in other domains, we developed a model transformation engine to perform in a distributed manner on a cloud. The solution is the adaptation of a model transformation tool for integration with MapReduce. Two architecturally distinct implementations are presented, one based on transformation rules and other based on model transformation procedures. The results are promising especially for the transformation of large and complex models.

CAPÍTULO 1

INTRODUÇÃO

Transformação de modelos é uma operação central em abordagens baseadas em *Model-Driven Engineering* (MDE), onde modelos são utilizados para descrever diferentes aspectos de um sistema em diferentes níveis de abstração [55]. Uma transformação de modelos é uma operação que recebe como entrada um ou mais modelos e gera como saída um novo conjunto de modelos [20]. Várias plataformas têm sido propostas e implementadas, fornecendo diferentes abordagens, paradigmas de programação e ferramentas para resolver tarefas comuns relacionadas às transformações de modelo, como por exemplo *Atlas Transformation Language* (ATL) [23], *Query/View/Transformation* (QVT) [30], *Epsilon Transformation Language* (ETL) [24] entre outras.

A crescente adoção de MDE para explorar e manipular grandes modelos evidenciou a escalabilidade como uma limitação crítica de várias ferramentas [57]. Grandes esforços em pesquisas já foram gastos no desenvolvimento de soluções para tentar superar essa limitação [52]. Modelos podem ser construídos, transformados e usados como base para a geração de códigos. O tempo para carregar modelos grandes é geralmente maior do que o tempo necessário para a verificação, fusão ou transformação dos mesmos [52] [11]. Assim como o tamanho dos modelos crescem, motores de transformação devem proporcionalmente evoluir seu desempenho. Há soluções que estudam maneiras de suportar grandes modelos [52].

Escalabilidade pode ser abordada de diferentes formas, como repositórios de modelo e operações de persistência, em especial, implementações de armazenamento em nuvem tem recebido grande atenção devido ao crescente interesse nos conceitos associados com o movimento *NoSQL*. Exemplo desses conceitos são encontrados em: (i) *Morsa*, abordagem que fornece acesso escalável para grandes modelos através de carga sob demanda [18];

(ii) soluções de persistência de modelos em bancos de dados baseados em grafos [3];(iii) e técnicas para evitar dependências entre modelo de dados e implementações de armazenamento em nuvem [10]. Linguagens de transformação atuais como ATL e QVT [30] que possuem arquiteturas centralizadas e baseadas na execução sequencial, não oferecem nenhum tipo de suporte para tratar de escalabilidade [34]. Outras soluções tentam demonstrar a importância da modularidade e encapsulamento para alcançar escalabilidade em MDE, ou seja, aspectos que afetam a modelagem e os recursos oferecidos pelos *frameworks* de modelagem [39].

Há soluções que exploram execução em paralelo, como Linda [8], uma linguagem de coordenação baseada em um conjunto de mecanismos básicos que permite transformações de modelos concorrentes. Existe também a implementação de um mecanismo de transformação paralelo para a linguagem ATL [56], que utiliza o conceito de *multi-threads*. Independente da estratégia de implementação estas abordagens e variantes continuam a ter como principal característica a arquitetura centralizada.

Como alternativa para resolver problemas de escalabilidade em grandes modelos, apresentamos a implementação de um motor de transformações executando de forma distribuída. O mecanismo foi construído através da adaptação do motor ATL padrão, para fazer chamadas ao *Framework* MapReduce [22], uma solução escalável para computação distribuída em grandes conjuntos de dados e assim validar sua aplicabilidade no contexto de transformações de modelos baseado na computação em nuvem. A escolha do ATL deve-se às características da linguagem e pelo fato de ser atualmente uma das ferramentas de transformação de modelos mais explorada no meio industrial e acadêmico.

Uma abordagem para aumentar o processamento de dados intensivos (como transformações de modelo) é fazer uso da distribuição [11]. Avanços recentes em computação em nuvem têm demonstrado em vários casos que podem ser uma solução para lidar com modelos muito grandes e outros problemas de transformação de modelos. Em um mecanismo de transformação com base na nuvem podemos ser capazes de distribuir o processamento, tanto para a análise de modelos de origem e de geração de modelos alvo [6].

Desenvolvemos duas implementações distintas arquiteturalmente, mas semelhantes funcionalmente, uma baseada em regras de transformação e outra baseada em procedimentos de transformação de modelos. A primeira tem como característica a utilização de duas fases. Na fase inicial são aplicados filtros por regras e na fase seguinte são executadas todas as instruções de processamento relacionadas a regra. A segunda implementação utiliza somente uma fase. Essa estratégia baseada em operações permite que o processamento seja realizado de uma forma independente obtendo melhores resultados.

Realizamos experimentos em um *cluster* para execução das transformações, e observamos que nossa solução trouxe resultados satisfatórios em relação a escalabilidade, sendo capaz de transformar grandes modelos.

Este trabalho está organizado da seguinte forma. No segundo capítulo destacamos o estado da arte e algumas bases para nossa solução. Trabalhos encontrados na literatura que se relacionam com o tema são comparados no capítulo 3. No capítulo quatro detalhamos a implementação das diferentes técnicas investigadas e comparamos os resultados da solução para escalabilidade em transformações de modelos. Por fim no capítulo 5 são expostas as conclusões e trabalhos futuros.

CAPÍTULO 2

ESTADO DA ARTE

Nesta seção, apresentamos uma síntese do estado da arte para modelagem, ferramentas de transformação de modelos, o paradigma de programação MapReduce e trabalho encontrados na literatura que tratam diretamente da questão de escalabilidade relacionada a modelos. Aqui fornecemos os conceitos e definições que descrevem os métodos considerados relevantes sobre nosso trabalho.

2.1 Modelos e Transformação de modelos

A evolução e a complexidade do desenvolvimento de software levaram ao longo das últimas décadas ao surgimento de técnicas e metodologias para facilitar o desenvolvimento de sistemas complexos [5]. *Engenharia Baseada em Modelos* (MDE) está relacionada com a concepção e especificação de linguagens de modelagem, mudando o foco de codificação para modelagem no processo de desenvolvimento de software [21].

MDE tem como hipótese base que qualquer artefato de software deve ser expresso por modelos, que são representados em diferentes níveis de abstração, representando o todo ou restringidos para tratar partes do sistema. Uma vez que modelos são todos interpretados por máquina, ferramentas podem automatizar (pelo menos parcialmente) um certo número de tarefas, como refatoração e refinamento de modelos (dos abstratos aos mais concretos), transformações ou a geração completa do código [21] [25] [36].

Apesar de não ser estritamente necessária, a construção de aplicações baseadas em modelos geralmente requer algum tipo de transformação [53]. Esta seção tem por objetivo apresentar as principais definições e terminologias (conforme literatura) relacionadas ao desenvolvimento baseado em modelos e conceitos inerentes a arquitetura de trans-

formações de modelos.

Definição 2.1 ***(Modelo)*****:** Um modelo é um artefato que representa um sistema, é formado por um conjunto de elementos de modelo [15]. Fornece uma representação simplificada de parte da funcionalidade, estrutura e ou comportamento de um sistema, transmite uma visão abstrata da realidade complexa. É escrito em uma linguagem de especificação e geralmente representado graficamente.

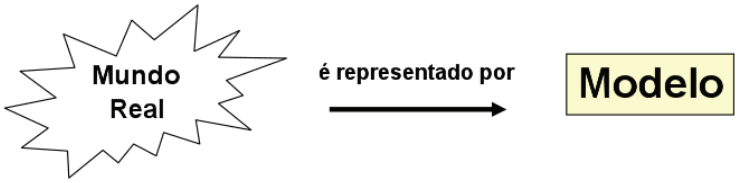

Figura 2.1: Visão abstrata representada por modelo.

Projetado para um propósito específico, permite usar um conceito sem ter que lidar com a sua complexidade do mundo real (Figura 2.1). Considere um complexo sistema de meteorologia, que recebe informações de um satélite para estimar previsões de tempo. Um modelo poderia ser utilizado para representar este sistema meteorológico e assim facilitar a compreensão de suas relações e funcionamento.

Definição 2.2 ***(Metamodelo)*****:** Um metamodelo é um modelo que define o tipo de elementos e relações de um modelo [15]. Contém um formalismo (sintaxe e semântica) preciso sobre os elementos de modelagem, relacionamentos e regras necessários para a construção de modelos.

Um metamodelo define as propriedades essenciais e características de um modelo, isso inclui o que significam e como se comportam. Um metamodelo é um modelo de tal forma que seu modelo de conformidade é um metametamodelo.

Definição 2.3 ***(Conformidade do modelo)*****:** A relação semântica entre um modelo e seu metamodelo [26]. Dizemos que um modelo M está de acordo com o seu metamodelo Mm ou é válido se forem satisfeitas e respeitadas as devidas condições e restrições.

Definição 2.4 ***(Metametamodelo)*****:** Um metametamodelo é um modelo que es-

pecifica a representação base para todos os modelos e metamodelos de um determinado contexto [15]. Esta em conformidade com seu próprio modelo, ou seja, está de acordo com ele mesmo.

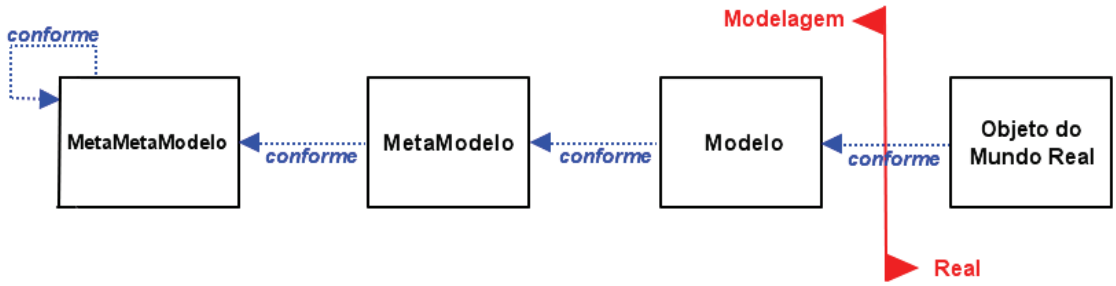

Figura 2.2: Níveis de modelagem.

A Figura 2.2 mostra um exemplo ilustrativo da arquitetura de três níveis, entre modelo, metamodelo e metametamodelo e suas respectivas relações de dependência.

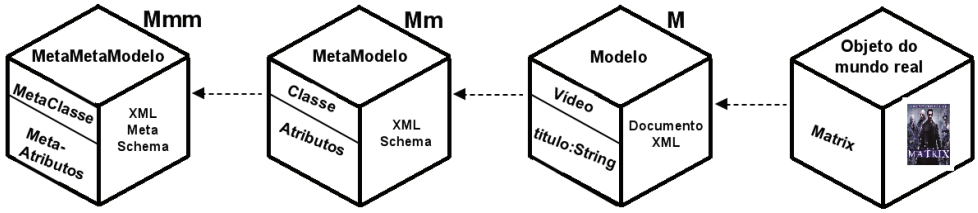

Figura 2.3: Exemplo de modelo, metamodelo e metametamodelo.

Na Figura 2.3 podemos observar um exemplo da organização entre modelo, metamodelo e metametamodelo. Como exemplo de objeto do mundo real temos um filme; sua representação modelada é mostrado no nível a esquerda, onde o modelo M descreve o conceito de vídeo com seus atributos (título no exemplo). O modelo está em conformidade com seu Metamodelo Mm, este descreve os conceitos utilizados em M para a definição do modelo (Classe e Atributo). Finalmente, o nível Mmm apresenta o metametamodelo que define os conceitos utilizados em Mm.

Definição 2.5 ***(Transformação de modelo)*****:** Uma transformação de modelo (M_t), constrói a partir de um conjunto de modelos de entrada (M_a) um novo conjunto de modelos de destino (M_b), de acordo com uma definição. Segundo [19], uma transformação de modelo tem as seguintes propriedades:

- está de acordo com um metamodelo de transformação;

- o corpo de transformação é criado tendo como valores os metamodelos de entrada ou saída;

- modelos de origem e destino são distintos;

- na execução de transformação, os elementos de entrada são comparados com base nos metamodelos de entrada;

- os elementos de saída são criados a partir da avaliação dos elementos correspondentes.

A Figura 2.4 contém um exemplo de transformação de modelo para modelo, onde um diagrama de classe (Modelo de entrada) escrito na notação UML [29] conforme a sintaxe da linguaguem (Metamodelo de entrada) é transformado em um diagrama de entidade-relacionamento (Modelo de saída) esse descrito em conformidade com seu metamodelo de saída.

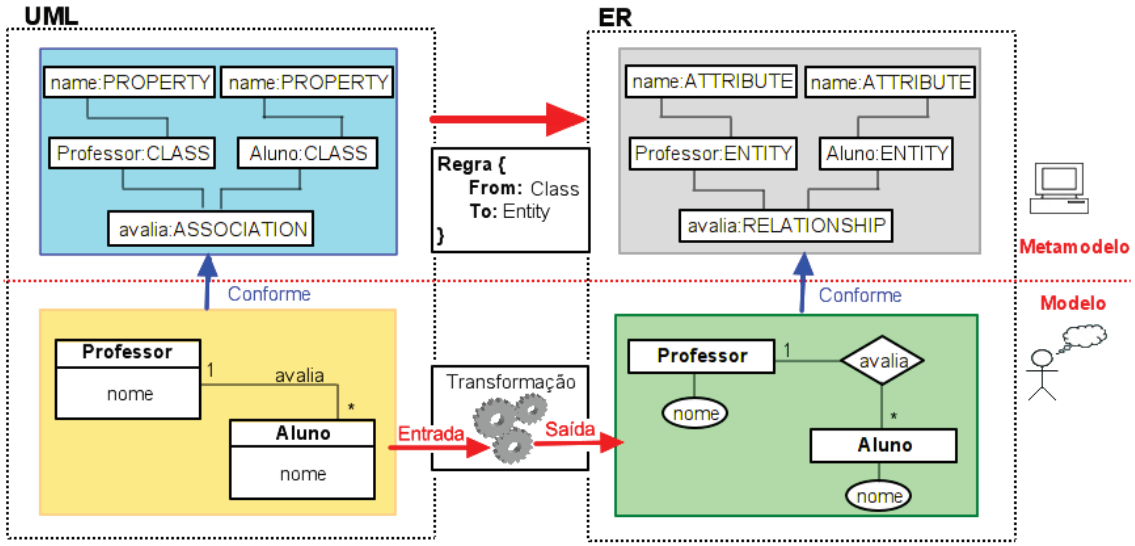

Figura 2.4: Exemplo Transformação de modelo UML para ER.

Definição 2.6 ***(Regra de transformação)*****:** é uma construção declarativa que contém um padrão de origem e destino. Descreve como um elemento do modelo de entrada é transformado em um elemento equivalente no modelo de saída [4].

Definição 2.7 ***(Motor de Transformação)*****:** um motor de transformação (T_e)

recebe um ou mais modelos como entrada (M_a1, ..., M_an) e uma regra de transformação que será interpretada (R_1), para assim produzir um ou mais modelos de saída (M_b1, ..., M_bn). Para executar uma transformação de modelo, um motor precisa executar os seguintes passos [4]:

i) Identificar os elementos no modelo fonte que precisam ser transformados;

ii) Para cada um dos elementos identificados produzir os elementos de destino associados;

iii) Produzir informações de rastreamento que ligam os elementos fonte e destino afetados pela regra. Uma visão global da arquitetura de um motor de transformação de modelo para modelo é apresentada pela Figura 2.5.

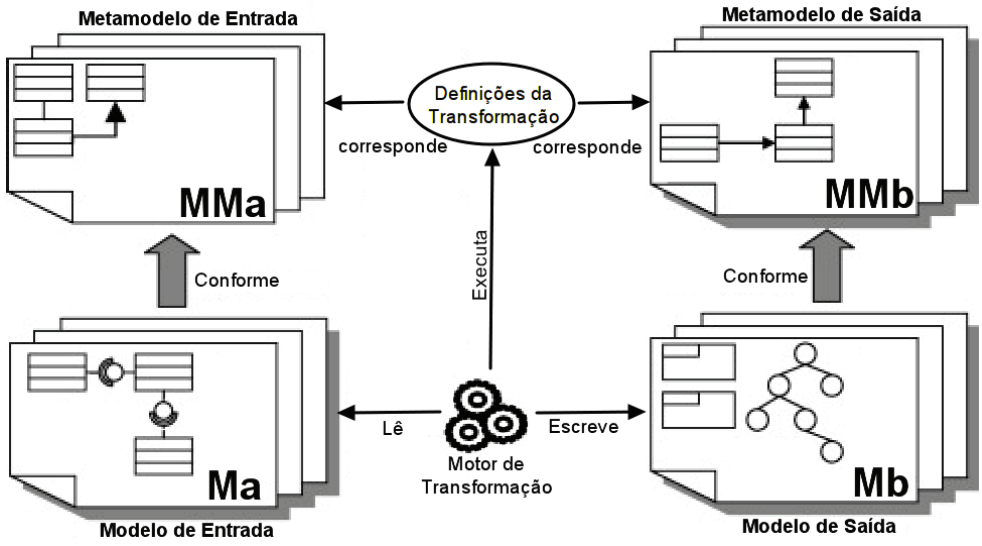

Figura 2.5: Visão do Motor de Transformação de modelo.

A Figura 2.5 mostra o cenário simples de uma transformação com um modelo de entrada Ma e um modelo de saída Mb. Ambos os modelos estão em conformidade com seus respectivos metamodelos MMa e MMb. As definições são executadas pelo motor de transformação, que lê os elementos do modelo de entrada e escreve os mesmos no modelo de saída de acordo com as regras de transformação.

2.2 *Frameworks* de transformação de modelo

Linguagens de transformação de modelo constituem um importante papel para a manipulação de modelos e permitem expressar transformações de maneira simples. Linguagens de diferente natureza têm sido propostas nos últimos anos, cada qual apropriada para um determinado tipo de tarefa de transformação. Abaixo uma lista com as linguagens de transformação de modelos encontradas com maior frequência nos trabalhos atuais.

Epsilon

A plataforma *Epsilon* contém um conjunto de linguagens específicas para tarefas relacionadas com o gerenciamento de modelos. Faz parte desta plataforma a *Epsilon Transformation Language* (ETL) [38], uma linguagem híbrida de transformação de modelo para modelo. ETL é utilizada para transformar um número arbitrário de modelos de entrada para um número arbitrário de modelos de saída, gerados por diferentes linguagens de modelagem e tecnologias a um nível elevado de abstração. Transformações ETL são organizadas em módulos. Um módulo pode conter uma série de regras de transformação, cada regra tem um nome único e também especifica uma origem e muitos parâmetros de saída [47].

QVT

Query/View/Transformation (QVT) [41] é a solução para transformações de modelo no âmbito de modelagem proposta pela OMG (*Object Management Group*). Ela é projetada para ser um padrão e não fornece uma implementação de referência. QVT é dividido em três sublinguagens, que juntas compõem uma linguagem de transformação híbrida, ou seja, aceita construções imperativas e declarativas. Transformações de modelo em QVT são expressos como relações entre modelos [47].

VIATRA

VIsual Automated model TRAnsformations (VIATRA) [13], é um *framework* para a definição e implementação de transformações no contexto de transformações baseadas

em verificação e validação. Fornece suporte para todo o ciclo de vida da engenharia de transformações de modelos, incluindo: especificação, projeto, execução, validação e manutenção de transformações para linguagens de modelagem e domínios da *Model Driven Architecture* (MDA). As regras são especificadas visualmente utilizando a notação UML. A partir destas especificações unidirecionais são derivados novos modelos [47].

GReAT

Graph Rewriting and Transformation (GReAT) [2] é uma linguagem de transformação de modelos para ferramentas de modelagem de domínio específico, é usada em conjunto com um ambiente de modelagem genérico (GME). É composta por três sub-linguagens: uma para especificação de padrões, outra para transformações de grafos e a última para controle de fluxo. Através da sub-linguagem GReAT permite a definição de padrões para grafos complexos e usa a noção de cardinalidade em cada vértice e aresta dos padrões. Para tratar as transformações em grafos, GReAT utiliza uma estratégia de unificação, onde os metamodelos do modelo de entrada e saída são usados para especificar explicitamente objetos temporários. Ao fim da transformação, os objetos temporários são removidos e cada modelo estará conforme seu próprio metamodelo. A terceira sub-linguagem surgiu da necessidade do usuário gerenciar a complexidade das transformações. Para tal, foi criada uma linguagem que dá ao desenvolvedor a liberdades de: sequenciar ou criar hierarquia nas regras de transformação, recursão e controle condicional do fluxo de execução.

Kermeta

Kermeta [45] é uma linguagem para especificação de modelos, metamodelos e transformações. Permite descrever tanto a estrutura como o comportamento de modelos. Utiliza o paradigma orientado a objetos (pacotes, classes, propriedades e operações). É uma linguagem imperativa, tendo sua sintaxe básica inspirada no Eiffel. As duas principais características da linguagem Kermeta são: a sua capacidade de extensão e a noção de tipificação de modelos. O primeiro recurso oferece mais flexibilidade aos desenvolvedores, permitindo-lhes facilmente manipular e reutilizar metamodelos existentes. O segundo é uma estratégia para a utilização de modelos que são subtipos ou variantes de um modelo

desejado no lugar dele, isto permite a detecção de erros de tipo no início do processo da transformação do modelo.

ATLAS Transformation Language

ATL (*ATLAS Transformation Language*) [1] é classificada como uma linguagem híbrida, por permitir construções declarativas e imperativas, do tipo *Model-to-Model* (M2M) [7]. É inspirada nos requisitos OMG QVT[1] e baseia-se no formalismo OCL[2] [32] [33]; a opção de utilizar OCL foi motivada por sua ampla aprovação em MDE e pelo fato de que é uma linguagem padrão apoiada pela OMG e pelos principais fornecedores de ferramentas. Transformações ATL são unidirecionais, operando sobre modelos de entrada que são somente de leitura e produzindo um novo modelo de saída que permite somente gravação. Sua sintaxe abstrata definida utilizando metamodelos, isto significa que cada transformação ATL é de fato um modelo, com todas as propriedades que lhe são inerentes [7].

ATL paralelo

Atualmente um mecanismo experimental de transformação paralela com ATL vem sendo testado e aperfeiçoado. A solução baseia-se no uso de *multi-threads*, a partir da hierarquia de operações geradas no processo de compilação das regras ATL, a execução é paralelizada com as sub-operações sendo separadas em diferentes *threads*, tendo como referencia a operação superior e partilhando entre si um mesmo conjunto de recursos.

Muitas outras abordagens para a transformação de modelos estão disponíveis no mercado e publicados na literatura, cada qual apropriada para diferentes cenários e soluções. Uma lista mais completa pode ser encontrada em [12] e [37].

As definições acima apresentam uma base formal, que permitem identificar importantes características da natureza das linguagens de transformação de modelos. Aspectos relacionados a funcionalidade, aplicação das regras, criação dos modelos graficamente

[1]Object Management Group (OMG) é um consórcio internacional que estabelece padrões na área de Orientação a Objetos.

[2]Object Constraint Language (OCL) é uma linguagem declarativa padrão para descrever regras em modelos UML.

ou por meio de programação, pontos comuns e variabilidades das abordagens de transformação de modelos existente.

Linguagens híbridas como ATL, QVT e ETL fornecem um esquema de execução declarativa baseada em regras, bem como características imperativas para lidar com cenários de transformação complexos. Já transformações com Kermeta não são baseados em regras, mas tem seu comportamento orientado por uma linguagem de ação. Epsilon e VIATRA são ferramentas predominantemente textuais, enquanto GReAT utiliza um ambiente de modelagem. Vários trabalhos têm propostas para a definição de características ideais e esquemas de classificação para linguagens de transformação [12] [44].

De todas as ferramentas investigadas para realizar transformações, escolhemos ATL. Assim, abordamos com mais detalhes esta linguagem na próxima seção.

2.3 Linguagem de Transformação ATL

ATL é atualmente uma das linguagens de transformações de modelo mais utilizadas, dentro da comunidade MDE, industrial e acadêmica. Além de código aberto, oferece um conjunto de ferramentas baseado no framework Eclipse. Estes são os motivos que levaram a escolha do ATL para nossa abordagem. Nas subseções a seguir são apresentados mais detalhes sobre a arquitetura ATL.

2.3.1 A linguagem ATL

Um programa de transformação ATL é composto de regras que definem a forma como os elementos do modelo de origem são combinados e navegados para criar e inicializar os elementos no modelo de destino. Cada regra é identificada por um nome único e podem ser especificadas em um estilo declarativo ou imperativo [33].

Transformações em ATL podem ser especificadas por meio de três tipos de regras: as regras combinadas, regras preguiçosas e chamadas a regras.

Regras combinadas - É o principal tipo de regra a ser utilizada em transformações declarativas, especifica qual o elemento de origem deve ser combinado, juntamente com o elemento de saída que deve ser gerado.

Regras preguiçosas - Semelhantes às regras combinadas, no entanto, elas não são executadas quando combinadas, mas sim, confiam em ser chamadas uma pela outra.

Chamadas a regras - São usadas para descrever transformações imperativas. Assim como uma regra combinada, pode gerar elementos no modelo de saída. No entanto pode ter parâmetros que são passados indicando como a regra é executada. A especificação de uma "regra chamada"será apresentada na próxima seção.

2.3.2 Motor ATL

Esta seção tem como objetivo apresentar algumas características do mecanismo de compilação e execução da linguagem ATL, necessários para processar uma transformação. Para este efeito, alguns termos comuns tais como operação, instrução e ambiente de execução são identificados e descritos.

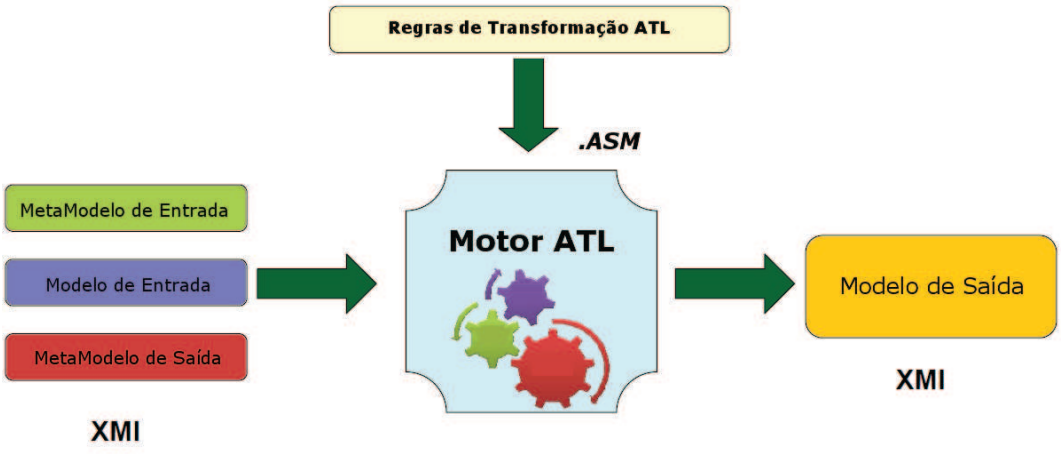

Figura 2.6: Arquitetura do motor ATL.

O motor ATL é implementado como uma máquina virtual (VM), a principal vantagem desta abordagem é a flexibilidade [31]. A Figura 2.6 apresenta um esboço da arquitetura do motor de transformação ATL, com modelo e metamodelos de entrada, a aplicação da

regras e por fim a geração do modelo de saída.

A fim de ilustrar esse processo, será utilizado como base uma regra ATL, ilustrada na Figura 2.8, que transforma modelos de classes para modelos relacionais. Onde classes declaradas como persistentes são transformadas em tabelas; as colunas da tabela e o tipo de dado são derivados dos atributos associados às classes; assim cada atributo é transformado numa única coluna (Figura 2.7).

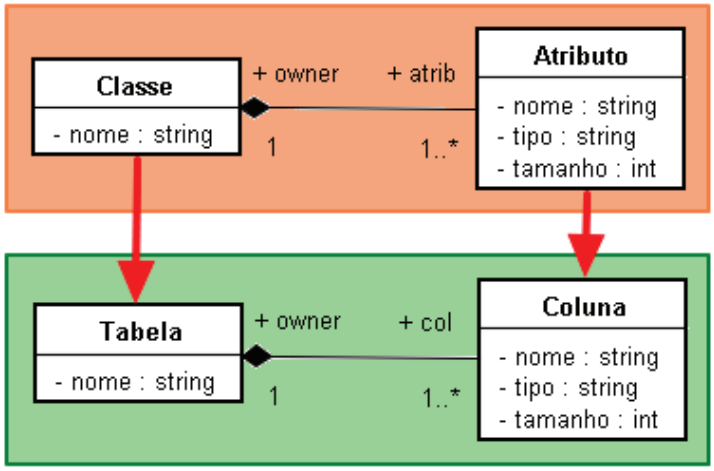

Figura 2.7: Modelos de entrada e saída para a transformação.

```
rule Classe2Tabela {
    from
        c : ClassDiagram!Classe
    to
        out : Relacional!Tabela (
            nome <- c.nome,
            coluna <- Set {atributos}
        ),
        atributos : Atributo!Coluna (
            nome <- thisModule.nome,
            tipo <- thisModule.tipo,
            tamanho <- thisModule.tamanho
        )
}
```

Modelo de Entrada

Modelo de Saída

Figura 2.8: Regra ATL para transformar Classes em Tabelas.

Definições

Algumas definições propostas por este trabalho, para entendimento dos principais elementos e componentes do motor ATL presentes na execução de uma transformação,

são apresentadas a seguir:

Definição 1. ***(Instrução)*:** Uma instrução é uma sentença escrita em uma linguagem de transformação de modelo a ser interpretada por um motor de transformação. Seja uma instrução I com o conjunto de parâmetros de entrada A que conduz a um resultado de B, então:

$$I(A) \rightarrow B$$

Onde B corresponde a comandos, como: $I(C) \rightarrow leia(C)$; $I(D, E) \rightarrow D = E * 2$; ou $I(F) \rightarrow imprima(F)$.

Definição 2. ***(Operação)*:** é uma estrutura ordenada de instruções, na seguinte forma:

$$Op = \{I_1(P_1), I_2(P_2), ..., I_n(P_n)\}$$

com n instruções, onde I_i é uma *i-ésima* instrução a ser interpretada e P_i é o conjunto de parâmetros da instrução I_i. Op é visto como um conjunto de instruções e seus respectivos parâmetros.

Um exemplo de operação Op pode ser $Op = \{I_1(C), I_5(D, E), I_8, (F)\}$, onde: $I_1(C) \rightarrow leia(C)$; $I_5(D, E) \rightarrow D = E * 2$; e $I_8(F) \rightarrow imprima(F)$.

Operações devem estar organizadas em uma hierarquia, esta é a forma de organizar um conjunto de operações para serem interpretadas por um motor de transformação. Uma ***(Hierarquia de Operações)*** tem a seguinte forma:

$$Ho = \{Op_1\{I_1\}, Op_2\{I_2\}, ..., Op_k\{I_k\}\}$$

com k operações, onde Op_i é a *i-ésima* operação a ser interpretada e I_i é o conjunto de instruções da operação Op_i.

$Ho = \{O_1\{I_1, I_5, I_8\}, O_2\{I_4, I_5, I_8\}, O_3\{I_2, I_3, I_4\}\}$

Este é o exemplo de uma hierarquia de operações. Ao longo do texto, para simplificar, uma hierarquia de operações pode ser interpretada como uma operação.

A execução completa de uma hierarquia de operações resulta na geração de um modelo de saída, esse processo é chamado de **execução da transformação**.

Definição 3. ***(Regra)*:** uma regra define como os elementos do modelo de entrada são combinados a fim de criar e inicializar os elementos do modelo de saída. Para este efeito, cada regra é definida de forma declarativa e tem a seguinte estrutura:

$$R = \{< name >, < match >, < exec >\}$$

onde $< name >$ é um nome único que identifica a regra; $< match >$ é um tipo de operação, que especifica os elementos de entrada que devem ser combinados; $< exec_i >$ é um tipo de operação que especifica o elemento de saída que será criado.

- **Ambiente de execução:** O ambiente de execução virtual contém todas as informações utilizadas por uma determinada execução, sendo recriado para executar cada operação. Espaço que oferece todos os recursos necessários para executar uma instrução, incluindo referências simbólicas para modelos, metamodelos e variáveis, uma hierarquia de operações, caracteres e valores associado ao estado de cada operação.

Para ser executado, o código ATL precisa ser compilado. Esta ação resulta num arquivo textual no formato denominado *asm*, o qual contém todas as informações necessárias para a construção de uma implementação da máquina virtual ATL. Assim, máquinas virtuais que visam executar transformações ATL, têm que ser capaz de ler o arquivo *asm* corretamente e realizar as operações que o mesmo especifica.

A execução de uma transformação inicia com uma operação principal (*main*), a qual engloba todas as demais operações, é responsável por inicializar o ambiente de execução para posteriormente controlar e gerenciar a ordem de chamada das mesmas. Para o exemplo da transformação de classes para tabelas, a hierarquia de operações é ilustrada na Figura 2.9.

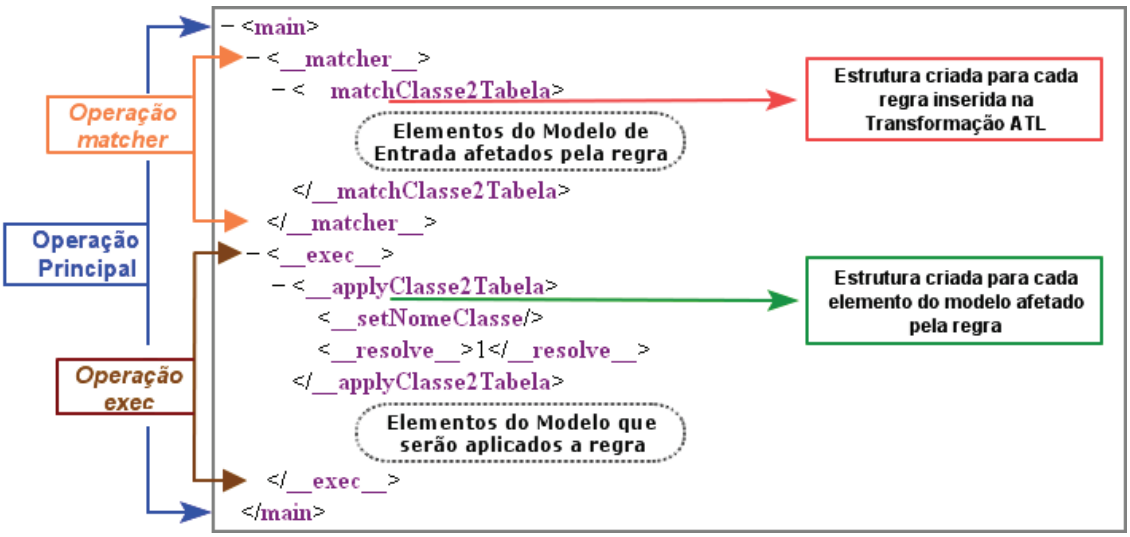

Figura 2.9: Hierarquia de operações criadas pelo ATL para a transformação.

A execução de um módulo ATL está organizado em fases sucessivas. No processo de transformação, a primeira atividade que o motor ATL executa é a inicialização dos recursos, nessa fase temos uma tarefa que combina os elementos do modelo de origem com as regras (*matching phase*). Em seguida, todas as inicializações (incluindo as ligações) são executadas de acordo com o algoritmo de resolução ATL. O algoritmo é chamado para cada ligação e transforma os elementos de entrada em elementos de saída.

O método responsável por executar as operação no ATL funciona de forma recursiva e similar a uma pilha. A execução consiste em descer até a base executando os cálculos ou rotinas de cada operação, então da base até o topo os resultados de cada operação são combinados resultando na transformação.

Resultantes do processo de compilação de um módulo ATL, as operações *matcher* e *exec* são responsáveis respectivamente por inicializar as ligações associadas a cada regra definida na transformação e posteriormente aplicá-las, assim são gerados os elementos no modelo de saída. Estas são distribuídas entre várias operações nomeadas de acordo com a regra, para o exemplo teríamos nessa ordem as operações *_matchClasse2Tabela* e *_applyClasse2Tabela*, como mostra a Figura 2.9.

Operação *match*

A operação match não tem parâmetros, percorre todos os elementos do modelo de entrada e para cada um deles, testa a condição especificada dentro do padrão da regra. Se o elemento satisfaz essa condição, a operação aloca um elemento vazio no modelo de saída. No exemplo, para cada elemento definido com Classe, a operação *_matchClasse2Tabela* alocada um elemento no modelo de saída do tipo Tabela.

Operação *exec*

A operação *exec* visa inicializar as propriedades dos elementos do modelo de saída, instanciados pela operação anterior. Captura explicitamente as relações entre os elementos de entrada e saída de uma regra. Para este efeito, a operação busca as propriedades do elemento no modelo de entrada e constrói novos valores que são atribuídos ao elemento no modelo de saída. No exemplo, temos associado a cada Classe um conjunto de Atributos (nome, tipo e tamanho), a operação _applyClasse2Table transformaria cada Atributo da classe em uma Coluna na Tabela de acordo com os parâmetros nome, tipo e tamanho.

2.3.3 Exemplo de Transformação ATL

Nesta seção é análisado uma tarefa simples de transformação ATL. O exemplo selecionado para análise é o *Book to Publication*, que pode ser encontrado no ATL zoo [23], onde estão disponíveis centenas de cenários de transformação.

O modelo de entrada (*Book*) contém informações sobre livros associado a um conjunto ordenado de capítulos (*Chapter*). O modelo de saída (*Publication*) contém informações sobre publicações genéricas. Para todos os livros, a transformação percorre cada capítulos deste livro e calcula a soma das páginas, para depois criar uma nova publicação e atribui os valores calculados.

Metamodelos

Cada livro (*Book*) é formado por um título e um conjunto de capítulos (*Chapter*), conforme ilustra a Figura 2.10 *a*. Cada capítulo tem um título e um número de páginas.

O metamodelo de saída (Figura 2.10 b) consiste de uma Publicação (*Publication*) que contém um título e o número de páginas.

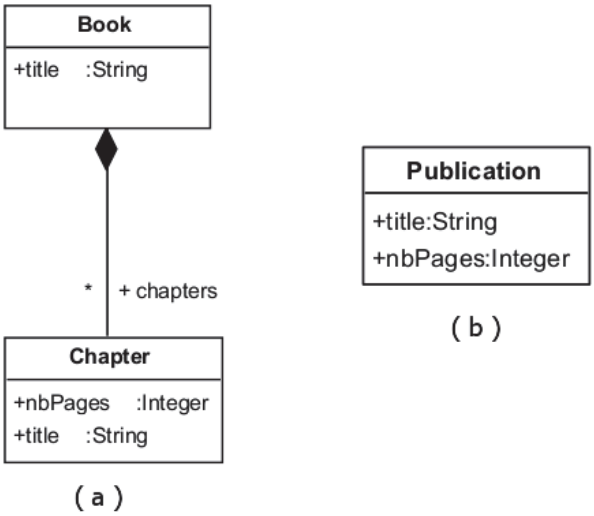

Figura 2.10: Metamodelos Book e Publication.

Especificação das regras

As regras para transformar um modelo *Book* em um modelo *Publication* são:

- Para cada instância *Book*, um exemplo de *Publication* tem de ser criado. Os atributos da instância *Publication* são definidos da seguinte forma:

- O título de uma publicação tem de ser definido com o título de um livro.

- O número total de páginas de uma publicação é a soma das páginas dos capítulos do livro.

Código ATL para o exemplo *Book to Publication*

O código ATL para a transformação de um modelo *Book* para um modelo *Publication* consiste da regra nomeada *Book2Publication*, como mostra o pseudocódigo abaixo (listagem 2.1). Esta regra usa um *helper* (expressões OCL utilizadas para definir variáveis e funções) em que todas as páginas de todos os capítulos de um livro são somadas.

Execução

Para executar o exemplo e verificar o resultado da transformação, é preciso instanciar o

```
module Book2Publication;
create OUT : Publication from IN : Book;

helper context Book!Book
  def : getSumPages() : Integer = self.chapters->collect(c|c.nbPages).sum();

rule Book2Publication {
  from
    b : Book!Book
  to
    out : Publication!Publication (
     title <- b.title,
     nbPages <- b.getSumPages()
    )
}
```

Listagem 2.1: Código ATL para o exemplo Book to Publication.

Listagem 2.2: Arquivo XMI para instanciar o modelo Book.

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:book="http://book.ecore">
  <book:Book title="Java"/>
  <book:Chapter numPages="10" title="Basic" author="Deitel"/>
  <book:Chapter numPages="15" title="Advanced" author="Jose"/>
</xmi:XMI>
```

modelo de entrada *Book* com os valores iniciais. O arquivo XMI (listagem 2.2) de entrada tem como valores o título, capítulos, o número de páginas e o autor.

Aplicando a regra *Book2Publication* para transformar um modelo *Book* em um modelo *Publication*, de acordo com nossos parâmetros de entrada, o resultado da transformação será um modelo de *Publication* tendo como valores o título "Java"e o número de páginas igual a 25, conforme mostra a Figura 2.11.

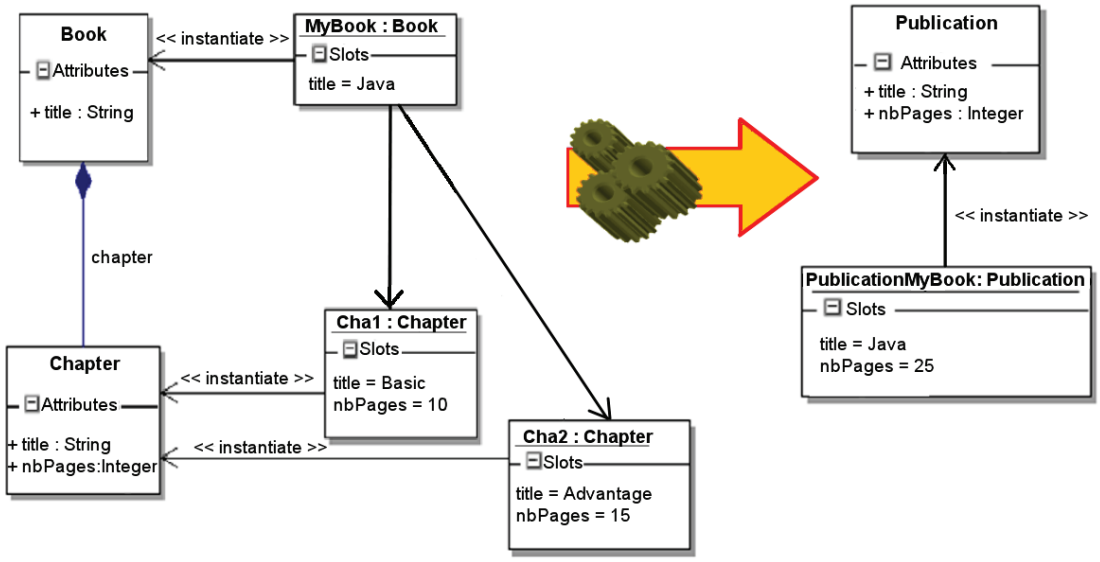

Figura 2.11: Resultado da transformação.

ATL é executado de forma centralizada, a invocação das operações na implementação atual do motor ATL ocorrem de forma sequencial, ou seja, a máquina virtual ATL executa as instruções de uma única operação por vez. Essa arquitetura apresenta problemas de escalabilidade, principalmente para transformar grandes modelos.

2.4 MapReduce

Nesta seção primeiramente serão abordados conceitos e informações sobre o modelo de computação MapReduce e na sequência apresentamos um exemplo com o objetivo de contextualizar e/ou fortalecer o entendimento básico da referida área.

2.4.1 Paradigma MapReduce

MapReduce é um paradigma de programação, inspirado nas funções *map* e *reduce* primitivas da linguagem de programação *Lisp*, que foi concebido para processar e gerar grande volume de dados [43] [42]. O sistema de execução automaticamente distribui a computação em larga escala de máquinas em *cluster*, como resultado tem-se um conjunto da dados sumarizados. Este modelo é passível de uma ampla variedade de tarefas do

mundo real, incluindo dados e aplicações com uso intensivo de computador, aprendizado de máquina, programação gráfica, programação multi-core, e assim por diante [14] [9].

A função *map* tem como entrada um único par de chave/valor, e produz como saída um número qualquer de novos pares de chave/valor. É fundamental que a operação *map* opere um par de cada vez. Isto permite a fácil paralelização, pois diferentes entradas podem ser processadas por máquinas distintas.

A função *reduce* converte todos os valores associados com uma única chave, e gera um conjunto de pares chave/valor com a mesma chave. Todos os mapeamentos precisam terminar antes da fase reduce começar. Uma vez que o redutor tem acesso a todos os valores com a mesma chave, pode realizar cálculos sequenciais com estes valores. Na etapa *reduce*, o paralelismo é explorado ao observar que chaves diferentes podem ser executados simultaneamente.

As funções tem respectivamente as seguintes assinaturas:

map: $(k1; v1) \rightarrow [(k2; v2)]$

reduce: $(k2; [v2]) \rightarrow [(k3; v3)]$

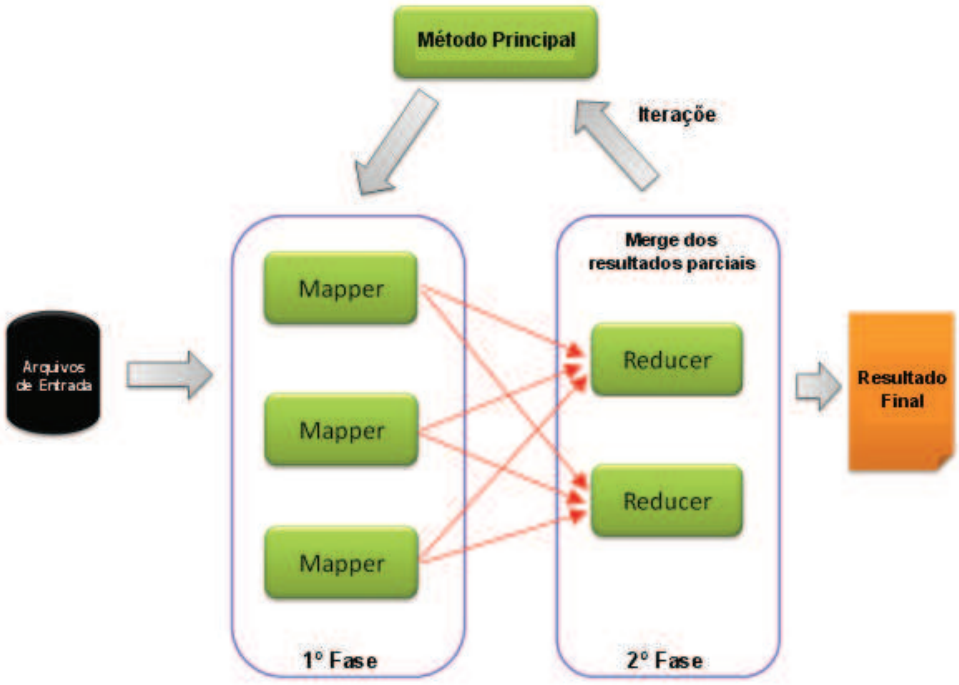

Figura 2.12: Fases do MapReduce.

A Figura 2.12 mostra o comportamento das fases do MapReduce e fluxo simplificado de execução do modelo. Um mapeador (Mapper) é um componente resultante da divisão do problema original e utilizado como um argumento para a função *map()*. Um redutor (Reducer) é o tipo de componente utilizado como um argumento para a função *reduce()*. É aplicado aos dados associados a uma chave específica e sempre produz dados com essa chave.

2.4.2 Exemplo implementação MapReduce

Vejamos o exemplo de um documento XML, no qual estão armazenadas as temperaturas máximas diárias registradas em determinada cidade e coletados em vários dias de medição. Queremos encontrar a temperatura máxima para cada cidade em todo o arquivo de dados.

Visão geral de Execução

O diagrama abaixo representado pela Figura 2.13 ilustra todo o cenário que será descrito no exemplo das temperaturas. A inserção e divisão dos dados de entrada, a função de mapeamento, a fase intermediária de ordenação dos dados, a passagem para a função de redução e por fim o resultado obtido.

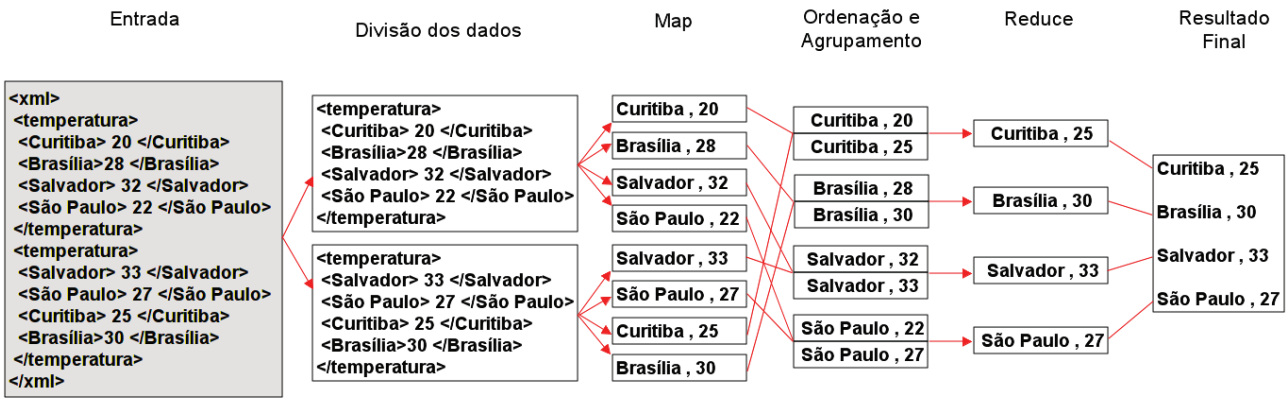

Figura 2.13: Visão geral de toda execução *Wordcount* no MapReduce.

A entrada para a aplicação é um arquivo XML, formado por *tags* (<*nome da cidade*>) que representam uma cidade e o conteúdo é a temperatura registrada naquela cidade no

```
classe MAPPER
função map(String chave, String valor)
INICIO
  //chave: nome da cidade
  //valor: todo o conteúdo entre as tags <cidade> e </cidade>

  PARA cada tag cidade do valor
      EMITIR(chave cidade, valor = temperatura);
  FIM PARA
FIM
```

Listagem 2.3: Pseudocódigo função map.

dia da medição. Uma aplicação real irá conter milhões ou até bilhões de registros, mas os princípios de chave e valor permanecem os mesmos.

Existem muitas possibilidades diferentes para traduzir estes registros em pares de [chave,valor]. Neste exemplo, a chave é o nome da cidade e a temperatura é o valor. Quando iniciamos o fluxo de trabalho, o *framework* divide a entrada em segmentos, passando cada segmento para uma máquina diferente. O arquivo XML é decomposto pelas medições diárias, utilizando a estrutura formada pela *tag* <*temperatura*>. Cada máquina executa a função *Map* na fração de dados que lhe é atribuída.

A classe *Map* (listagem 2.3) recebe as linhas de texto (os arquivos de texto são divididos em linhas) e quebra em palavras. A saída para cada palavra é um tupla (string, int), na seguinte forma (”palavra”, 1), uma vez que cada tupla corresponde à primeira ocorrência de cada palavra, assim a frequência inicial de cada palavra é 1. Emitido os pares de [chave, valor] estes são embaralhados (*shuffled*), significa que os pares com a mesma chave são agrupados e passados para uma única máquina, que irá executar a função *Reduce* sobre eles.

A função *reduce* tem uma coleção de pares [chave, valor] e irá agregá-los de acordo com o especificado na função. A classe *Reduce*(listagem 2.4) recebe uma coleção de dados de entrada na forma [($cidade_1$, $temperatura_1$), ($cidade_1$, $temperatura_2$),...] em que a cidade é a mesma, mas com diferentes temperaturas. Estas coleções são o resultado de um processo de separação que é parte integrante do MapReduce e que reúne todos os dados com a mesma cidade em um conjunto. A função *reduce* reúne os dados dentro de

```
classe REDUCER
função reduce(String chave, <Lista de temperaturas> lista,
INICIO
  //chave: cada cidade do arquivo XML
  //lista: lista com todas as temperaturas encontradas para esta cidade
    int maior_temperatura = 0;
    PARA cada temperatura t = chave na lista
    {
      SE t > maior_temperatura
     ENTÂO
       maior_temperatura = t;
      FIM SE
    }
    EMITIR(cidade, maior_temperatura);
  }
FIM
```

Listagem 2.4: Pseudocódigo função reduce.

um *DataNode*[3] e também reúne o resultado dos diferentes *datanodes* em uma coleção final, onde as cidades agora são únicas, já com a maior temperatura encontrada.

2.5 Armazenamento e *NoSQL* em Transformação de Modelos

Modelos podem ser armazenados utilizando diferentes abordagens de persistência [18]. Estas soluções são arquivos regulares (XMI), bancos de dados relacionais através do mapeamento objeto-relacional ou um nível mais alto de abstração, os chamados repositórios de modelos [44] [49].

Para lidar com o crescimento do tamanho dos modelos surgiram algumas abordagens, tais como Teneo[4] e CDO[5], para persistência de modelos em bancos de dados. Estes são dois projetos EMF (Eclipse Modeling Framework) [51], focados na persistência de modelos e metamodelos usando mapeamento objeto-relacional, o que lhes permitem serializar modelos em bancos de dados relacionais. Embora útil, na prática, estas abordagens parecem ser soluções temporárias que tentam compensar a falta de encapsulamento e construções modularidade em linguagens de modelagem [16]. A questão a ser tratada no longo prazo, não é como gerenciar grandes modelos monolíticos, mas como separá-los em pequenos

[3]O DataNode é responsável por realizar o gerenciamento de dados.
[4]Teneo é uma solução de persistência relacional para o Eclipse Modeling Framework (EMF).
[5]CDO (Conect Data Objects) é um repositório distribuído para modelos e metamodelos EMF.

modelos modulares e reutilizáveis.

Atuais ferramentas de suporte a persistência de modelos através de arquivos XMI apresentam uma série de inconvenientes [17]: (i) suporte textual baseado em XML não é escalável em termos de uso de disco, (ii) uso excessivo de memória ao carregar grandes modelos, (iii) apoio inadequado para acesso simultâneo e controle de versões, que são essenciais para o desenvolvimento em equipe.

Soluções tradicionais de persistência

Uma visão geral do estado atual da prática de persistência de modelo e das abordagens propostas para gerir modelos em grande escala de forma eficiente é apresentada em [3], também é fornecida uma análise comparativa entre as tradicionais soluções apoiadas por bancos relacionais e um novo paradigma dos bancos orientados a grafos. Os bancos de dados NoSQL surgiram como uma alternativa promissora que ultrapassa algumas das limitações de bases de dados relacionais para persistência de modelos em larga escala. Neo4j [46] e OrientDB [54] tentam resolver os problemas de escalabilidade usando bases de dados orientadas a grafos para armazenar grandes modelos. O raciocínio por trás da escolha dessas tecnologias foi que Neo4j é um banco de dados de grafos extremamente difundido e estável, enquanto OrientDB não só fornece uma camada de documento e uma camada de grafos, mas também tem uma licença flexível. Protótipos integrando Neo4j e OrientDB com EMF foram implementadas e demonstram resultados de desempenho que ultrapassam o armazenamento de arquivos XMI bem como soluções baseados em Teneo. Estes resultados podem promover o aprofundamento da investigação e o desenvolvimento de soluções de grande escala para persistência de modelos baseado em banco de dados NoSQL que utiliza grafo como forma de representação dos dados.

Morsa

Morsa [18] é uma abordagem baseada em NoSQL, fornece acesso escalável para grandes modelos através de carga sob demanda, capaz de carregar totalmente grandes modelos com uma quantidade limitada de memória. Este mecanismo se baseia em um cache

de objetos, com uma política configurável, que decide se o cache está cheio ou não e quais objetos devem ser descarregados, assim reduz consultas de dados e gerencia o uso de memória. Morsa conta com um banco de dados NoSQL (MongoDB) para armazenar modelos EMF como coleções de documentos. Cada elemento do modelo é armazenado por documento, tendo seus atributos armazenados como um par de chave/valor juntamente com seus metadados. Morsa é uma solução de persistência de modelos que visa atingir escalabilidade no acesso de grades modelos.

EMF fragments

Eclipse Modeling Framework (EMF) e *Map/Reduce* podem ser combinados utilizando às vantagens de ambas as tecnologias, é o que propõem [48]. A abordagem consiste em duas partes: o armazenamento de grandes modelos EMF em um sistema distribuído baseados no armazenamento chave-valor (HBase Hadoop) e em seguida, construir programas *Map/Reduce* que usam APIs EMF para processar esses grandes modelos EMF. O *framework EMF fragments* é uma camada de persistência para EMF, que foi projetado para armazenar grandes modelos de dados orientados a objetos. EMF fragments enfatiza o armazenamento rápido de novos dados e a navegação rápida de modelos persistidos. EMF fragments é diferentes dos *frameworks* baseados em mapeamento objeto relacional (ORM), enquanto ORM mapeiam individualmente objetos, atributos e referências para entrada de banco de dados, EMF fragments mapeia fragmentos maiores de um modelo para URIs. Isso permite armazenar modelos em uma ampla gama de repositório de dados distribuídos incluindo bancos de dados NoSQL como o MongoDB ou HBase. EMF fragments, portanto, fornece armazenamento para estruturas de dados tipificadas que permitem a análises com base em *Map/Reduce*, ou seja, para a computação em nuvem. O *framework EMF fragments* permite fragmentação transparente e automatizada de modelos. Fragmentos são gerenciados automaticamente: quando você cria, apaga, move ou edita os elementos do modelo, novos fragmentos são criados e os elementos distribuídos na nuvem.

Comparação entre soluções

Montamos um quadro onde são abordados distintos aspectos, apresentados em um nível mais conceitual, ou seja, uma comparação não arquitetural. O foco é sobre os principais componentes, características, ferramentas de execução e como eles estão relacionados. A motivação para ter uma visão relacionada são as possíveis soluções que podem ser compostas utilizando-se as vantagens e desvantagens de cada uma.

Tabela 2.1: Quadro comparativo entre soluções.

	Morsa	**EMF MapReduce**	**Neo4J e OrientDB**	**EMF e CDO**	**ATL e QVT**	**Epsilon**
Armazenamento local	não	não	não	não	-	-
Armazenamento baseado em nuvem	sim	sim	sim	sim	-	-
Estratégia de fragmentação	não	sim	não	não	não	não
Estratégia de cache	Política de substituição de cache	não	não	não	Lazy evolution and caching	suporte atráves do uso explicito de anotações
Carregamento sob demanda	uma a um ou carga parcial	recursos gerenciados em background	Índice contendo os IDs	carregamento parcial	Modelos e Metamodelos precisam ser carregados explicitamante	Carga e validação automática dos modelos e metamodelos
Tipo de banco de dados	MongoDB	Hbase	Graph Database	ORM	-	-
Necessidade de modelos/classes especificas	não	não	-	requer préprocessamento dos metamodelos antes de persistir	links de rastreamento	Rastreabilidade de requisitos e refinamento
Criação/gerenciamento automático de recursos	sim	sim	Apenas informações do modelo são armazenados na totalidade	não é uma aplicação transparente	não	necessita de implemantação na linguagem própria
Estratégia de navegação para modelo	Objetos necessários	Objetos necessários	Transversal (conexões diretas)	sob demanda em memória	cargas parciais	expressões lógicas
Suporta transformações bidirecionais	-	-	-	-	ATL-não QVT-sim	não
Capacidade de lidar com modelos grandes/complexos	sim	sim	sim	não	não	não
Abordagem para lidar com modelos EMF grandes	mapeamento para banco de dados não-relacionais	armazenamento distribuído e mapeamento de fragmentos	Algoritmos matemáticos para travesia	recurso pode ser um arquivo ou uma entrada no banco de dados	programação imperativa	algoritmos de transformações conectáveis (comparação)

Como pode ser observado a partir da tabela 2.1, onde organizamos diferentes abordagens para persistência e transformação de modelos, as relações entre as abordagens são complexas. Sendo as mesmas equivalentes em algumas categorias e diferentes em outros aspectos. Isto é devido à arquitetura relativamente complexa que compreende cada

solução.

Soluções para persistência de modelos (Morsa, EMF MapReduce, CDO, Neo4J e OrientDB) enfatizam estratégias para armazenamento distribuído, técnicas de carregamento e gerenciamento automático de recursos, ou seja, estão preparadas para tratar de grandes modelos. Enquanto que as soluções de transformações (Epsilon, ATL e QVT) focam em estratégias de cache e carregamento sob demanda, fortemente dependentes do uso de memória e não evoluíram para suportar modelos grandes e complexos.

Estratégias de navegação dos modelos também são exploradas de diferentes formas pelas soluções de armazenamento, como conexões transversais e diretas, o que culmina na maior interdependência entre modelos e elementos. Já os frameworks de transformação aderem a recursos de programação e algoritmos. Outra lacuna que identificamos é a falta de componentes para integração entre soluções de persistência e ferramentas de transformações.

Os resultados do nosso estudo mostraram que, em um nível conceitual, é possível ter interoperação entre técnicas, soluções e abordagens existentes com base em transformações do modelos.

2.6 Sumário

Escalabilidade é uma propriedade desejável em *Model-Driven Engineering* (MDE) e um gargalo para aplicações industriais. Problemas de escalabilidade surgem quando modelos grandes (da ordem de milhares de elementos) são usados em processos de MDE [3] [40]. O foco atual das pesquisas em MDE estão em linguagens declarativas para gestão de modelos e em mecanismos escaláveis para persistir modelos (por exemplo, utilizando bases de dados) [3].

Estes problemas podem ser divididos em três categorias [3] [28]: (i) Persistência do modelo: o armazenamento de grandes modelos, a capacidade de acessar e atualizar tais modelos com pouca memória e tempo de execução. (ii) Consulta e transformações de

modelos: capacidade de realizar consultas intensivas ou complexas e transformações em grandes modelos, com baixo tempo de execução. (iii) Trabalho colaborativo: vários desenvolvedores verificam uma parte de seu modelo, consultam ou editam, além de serem capaz de submeter as suas mudanças com sucesso.

Neste trabalho nosso foco está sobre a segunda categoria de problemas, que são as transformações de modelos. O tamanho e a complexidade dos modelos de entrada são fatores diretos que afetam o desempenho de execução de transformação. Aumentando o número de elementos de entrada, também aumenta o número de elementos que são potencialmente cobertos por regras. Isto resulta em um aumento dos elementos a ser transformado e no tempo geral de execução.

Transformação de modelo é um domínio de aplicação onde a otimização da velocidade com base na execução distribuída tem um grande potencial, especialmente no caso de grandes modelos industriais, não escalares e com grandes quantidade de regras. As ferramentas raramente exploram execução distribuída para melhorar, tanto em termos de velocidade de execução como de escalabilidade. E mesmo variantes de implementação que usam ATL para executar transformações em paralelo continuam a focar na execução centralizada.

CAPÍTULO 3

TRANSFORMAÇÃO DE MODELOS BASEADA EM MAPREDUCE

Nesta seção apresentamos duas formas de implementar um motor de transformação integrado com MapReduce executando na nuvem, uma baseada em regras e outra organizada por operações ATL, na sequência descrevemos os cenários utilizados para os testes e ao fim observamos a real compatibilidade de unir transformação de modelo e computação distribuída através de ATL e MapReduce.

Primeiramente identificamos o ponto chave na execução do motor de transformação ATL que correspondente ao sequenciamento e controle das operações. Alteramos o mecanismo para implementar uma chamada ao *Framework* MapReduce, a paralelização pode ser alcançada através da divisão das operações que podem ser processadas individualmente, simultaneamente e em menos tempo do que a forma sequencial. Ao final, os dados oriundos dos diversos processamentos devem ser agregados para obter o resultado final.

Hadoop [22] é uma implementação *open-source* de MapReduce, de grande extensão e madura vem sendo usada com sucesso em muitas aplicações para processamento de grandes quantidades de dados. Seu potencial é explorado tanto no meio industrial como pela comunidade científica acadêmica.

3.1 Execução de transformação distribuída baseada em regras

A implementação baseada em regras tem como característica a utilização das duas fases do MapReduce. Na função de mapeamento são aplicados filtros por nome de regras, assim as operações são agrupadas com base no nome. Na fase seguinte são executadas

todas as instruções e sub-operações relacionadas a regra.

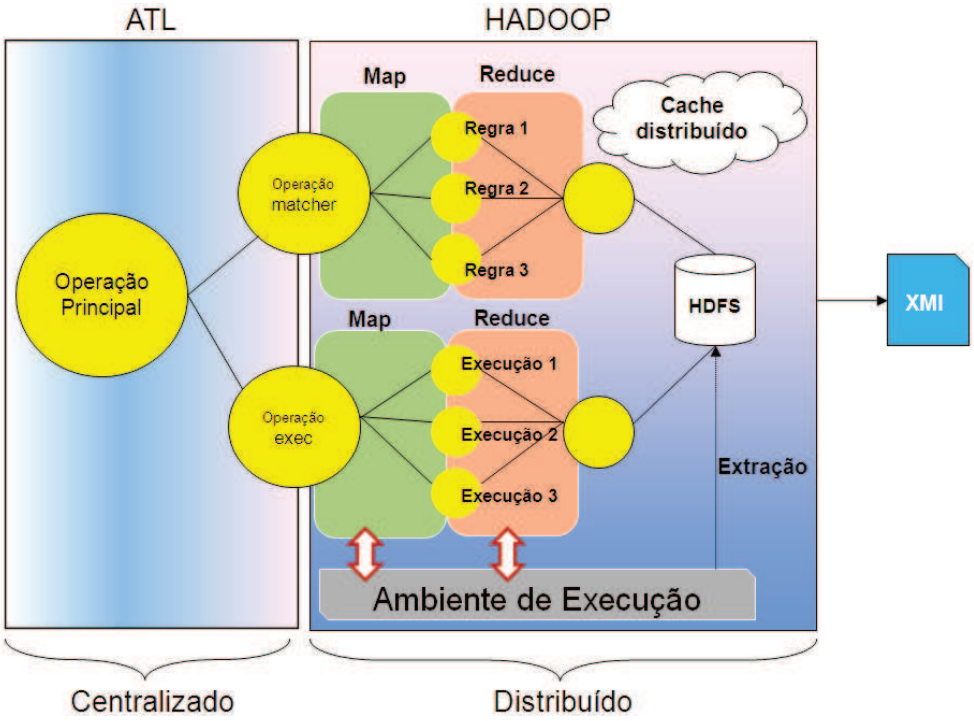

Figura 3.1: Arquitetura de execução ATL com Hadoop baseada em regras.

A Figura 3.1 demostra a forma como operações que compõem uma transformação são distribuídas em termos de processos executados em uma nuvem. A operação principal continua a ser executada de forma centralizada, exercendo um controle sobre todas as demais operações pertencentes a uma determinada transformação. A seguir apresentamos passo a passo como é realizada a transição da execução centralizada para distribuída, o carregamento dos modelos, a aplicação de cada uma das funções MapReduce e a extração do modelo resultante da transformação para a abordagem baseada em regras.

1.Transição da transformação

A transição é realizada em duas fases sucessivas (*matcher* e *exec*), visto que há uma dependência entre elas:

1.1 Distribuição do *matcher*

Num primeiro passo, cada operação *matcher* é serializada e enviada ao Hadoop. A

execução inicia com a leitura do arquivo serializado armazenado no HDFS[1]. Normalmente as entradas para o Hadoop vêm de um conjunto de arquivos carregados no *cluster* do HDFS, mas não necessariamente isso é obrigatório, podemos usar um diretório ou algum arquivo padrão para especificar a entrada de tarefas para o MapReduce.

1.2 Distribuição do *exec*

Posteriormente a conclusão da operação *matcher*, o controle retorna a operação principal do ATL que serializa a operação *exec*, envia para o HDFS e executa as mesmas funções sobre esta.

2.Desmembramento do modelo

Como o modelo se faz necessário para executar cada uma das operações do ATL submetidas ao *cluster*, e estes são arquivos muito grandes, desenvolvemos um formato de entrada personalizado para ler o arquivo XMI específico do nosso modelo.

Na estrutura que captura os arquivos de entrada, definimos como os modelos de entrada são lidos e divididos. Ao iniciar nossa transformação no Hadoop, um parâmetro é passado com o caminho que contém os arquivos para leitura. A estrutura irá ler o modelo neste diretório, em seguida dividir esse arquivo em um ou mais *InputSplits*. Um *InputSplit* descreve uma unidade de trabalho, um conjunto de dados chamado *job*, o qual em nossa implementação é composto de um pedaço completo do modelo de entrada.

Por padrão o Hadoop quebra um arquivo em pedaços de 64MB (o mesmo tamanho de blocos no HDFS), mas é possível controlar este valor, uma vez que alguns formatos de arquivos não são passíveis de processamento em partes. Ao escrever um *InputFormat* personalizado, controlamos como o arquivo é quebrado ou dividido. Assim garantimos que o trecho do modelo de entrada capturado é a representação completa de um modelo. Para essa primeira implementação, optamos por blocos com aproximadamente 200Mb.

O método (listagem 3.1) é invocado repetidamente sobre o modelo de entrada até o

[1]Hadoop Distributed File System (HDFS) é projetado para armazenar grandes conjuntos de dados de forma confiável e transmiti-los com alta largura de banda [50].

```
1  função inputFormat(Modelo m)
2  INICIO
3      tamanho = m.size();
4      maximo = 100Mb;
5      SE (tamanho < maximo)
6        bloco = m;
7      SENÃO
8        inicio = 0;
9        PARA(j=0 até j < tamanho)
10      bloco = getFileBlock(m, inicio, tamanho);
11      inicio = inicio + 100Mb;
12       FIM PARA
13     FIM SE
14 FIM
```

Listagem 3.1: Método para leitura e divisão do modelo de entrada.

modelo inteiro for consumido. Cada invocação do método de leitura leva a outra chamada para o método *map()* da classe que implementa o mapeamento. Ao processar um arquivo em pedaços, permitimos que várias tarefas *map* operem um único arquivo em paralelo.

3.Mapeamento por regra

Após a leitura de um trecho do modelo, o processamento então é dividido em duas fases distribuídas (*map* e *reduce*). A primeira decompõe as operações *matcher* e *exec*, criando pares de chave/valor, onde o nome das regras {*Rule1, Rule2, Rule3, ...*} / {*Apply1, Apply2, Apply3, ...*} são a chave e o valor é todo o conteúdo de cada sub-operação associada a regra, ou seja, arquivos binários contendo instâncias que representam as operações.

O método *map* recebe como parâmetros uma sequência de *bytes* e retorna como chave um texto (regra) e como valor um objeto serializado (operação). O pseudocódigo da função *map* é apresentados na *listagem 3.2*. Assim, o método map define cada nome de regra como chave e o valor é a operação correspondente a essa regra, que se encontra serializada e pronta para execução.

A Figura 3.2 ilustra a passagem entre as etapas de desmembramento do modelo e mapeamento. O principal ponto nesse processo é a geração das unidades de trabalho e a organização das mesmas pelo nome da regra.

```
1  função map(Regra key, OperacaoATL value)
2  INICIO
3      operacao_map = deserializar(value);
4      Lista lista = operation_map.getInstructions();
5      PARA(j=0 até j < tamanho da lista)
6        Bytecode bytecode = lista.get(j);
7        nova_operacao = ambiente_execucao(bytecode)
8        SE(nova_operacao for uma instancia de ASMOperation)
9       Cria um Map passando (nova_operacao.getNome() e nova_operacao)
10       FIM SE
11     FIM PARA
12 FIM
```

Listagem 3.2: Método map.

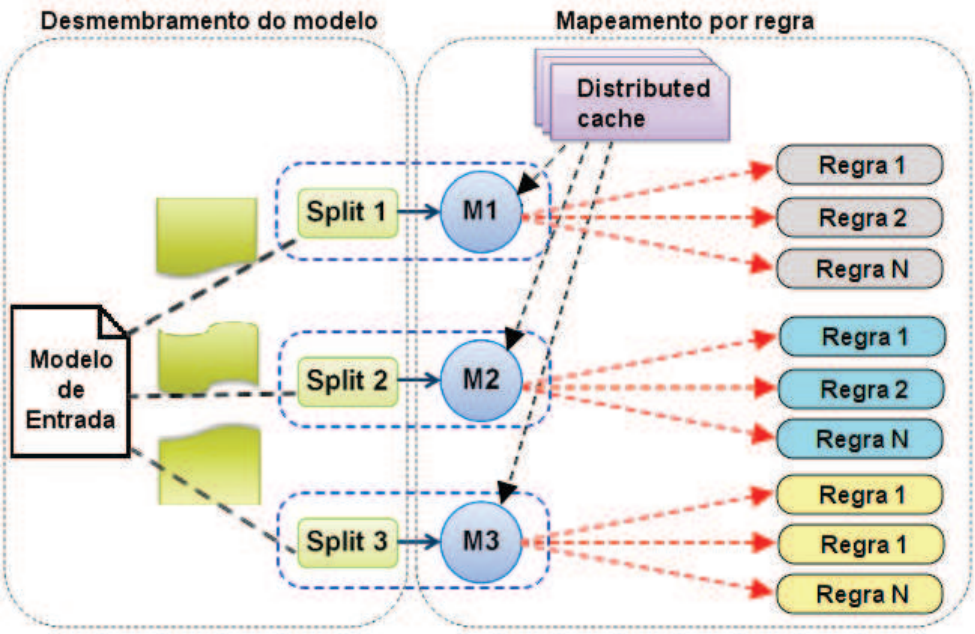

Figura 3.2: Mapeamento da transformação por regra.

O processo de transferência de saídas do *map* para os redutores é conhecido como *shuffling*. Um subconjunto diferente de cada chave é atribuído a cada nó redutor; esses subconjuntos (partições) são as entradas para as tarefas de redução. Cada tarefa de mapeamento pode emitir pares de chave/valor para qualquer partição, todos os valores para a mesma chave (regra) são sempre reduzidos juntos, independentemente de qual é a sua origem. Portanto, os nós de *map* devem concordar sobre para onde enviar as peças diferentes dos dados intermediários.

Hadoop fornece um mecanismo chamado *cache distribuído (DistributedCache)* [58],

o qual possibilita que arquivos de dados necessários para a inicialização ou bibliotecas de código que precisam ser acessados em todos os nós do cluster estejam disponíveis ou visíveis.

4.Execução das operações

Na segunda fase (reduce) as sub-operações são executadas seguindo suas instruções específicas definidas no ATL, ou seja, tudo que o motor ATL executava de forma sequencial aqui é paralelizado. Porém, como a execução das sub-operações é dependente das variáveis disponíveis no *Ambiente de Execução*, se fez necessário disponibilizar uma instância única para todos os nós envolvidos com a operação.

Utilizando-se do mecanismo de *cache distribuído*, os metamodelos de entrada e saída juntamente com o arquivo *asm* compilado necessários para a transformação e de pequeno porte são passados ao Hadoop, de posse deste é criada uma instancia única. Assim o ambiente de execução ATL permite o carregamento ou recarregamento das operações em sua tabela de pesquisa. Os resultados de cada operação são inseridos num modelos de saída único, que a partir de então concentra-se no lado Hadoop.

Ao distribuir as operações de transformação no *cluster* para execução, criamos a dependência de que todas executem sobre um mesmo *ambiente de execução*. Como o motor ATL em suas fases iniciais, cria a correspondência entre os elementos de entrada e saída, inicializando todos os recursos necessários para o processo de transformação, capturamos esse estado para criar uma instancia única do ambiente de execução e distribuir esta para todos os nós do *cluster* envolvidos na transformação através do *cache distribuído*. Nota-se que, nesta fase, os elementos do modelo de destino são simplesmente alocados. Assim os nós estão preparados para executar as operações da transformação.

O método *reduce* que recebe os valores vindos do mapeamento, executa as operações ATL e deposita os resultados nas variáveis globais do *ambiente de execução* é apresentado pelo pseudocódigo *listagem 3.3*. Tanto para o *matcher* como para o *exec* essas operações representam as instruções especificas de cada regra sendo executadas.

```
1  função reduce(Text regra, Iterator<OperacoesATL> operacao)
2  INICIO
3    ENQUANTO(existirem Operações)
4      operacao_reduce = deserializar(operacao);
5      SE(operacao_reduce !=null && ambiente_execucao!=null){
6         ASMOperation resultado = Chamar API do ATL (ambiente_execucao, operacao_reduce);
7      SENÃO
8         lançar Exceção;
9      FIM SE
10
11     ambiente_execucao = atualizacao(resultado);
12
13     returna (Chave regra, Valor resultado da operacao_reduce);
14   FIM ENQUANTO
15 FIM
```

Listagem 3.3: Método reduce.

As saídas intermediárias da fase de mapeamento, a execução das operação de cada regra e a produção do modelo resultante da transformação são representados na Figura 3.3.

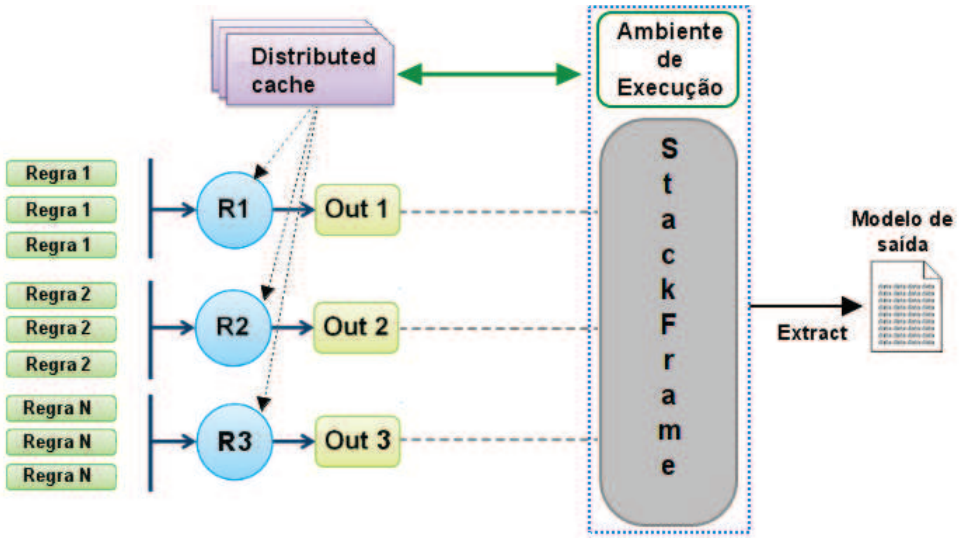

Figura 3.3: Execução da transformação por regra.

5.Retorno / Modelo resultante Ao fim do processamento da operação *matcher*, o controle volta então a operação principal, que serializa a operação *exec* e passa para o Hadoop iniciando um novo ciclo. Por fim, ao término de ambas operações, o modelo resultante da transformação é extraído no HDFS do Hadoop, utilizando a própria API disponibilizada pelo ATL.

3.2 Execução de transformação distribuída baseada em operações

No decorrer deste estudo, observando a implementação baseada em regras, identificamos a possibilidade de estabelecer outra forma de distribuir as transformações. Implementações com MapReduce não necessitam obrigatoriamente de uma fase *reduce*, pode-se utilizar somente da tarefa de mapeamento. Essa é a estratégia aplicada na solução baseada em operações. A operação principal é dividida para executar cada uma de suas sub-operações em mapeadores diferentes. O processamento de cada operação é realizado de forma independente e não há necessidade de consolidar os resultados individuais ou agregados de cada operação, pois o processamento de cada transformação já foi atribuído diretamente no modelo de saída.

A Figura 3.4 ilustra a forma como uma transformação é distribuída para essa abordagem. A operação principal inicia a execução de forma centralizada, porém a mesma é serializada e todo o restante do processamento ocorre em paralelo. A seguir apresentamos as etapas desse segundo modelo de implementação.

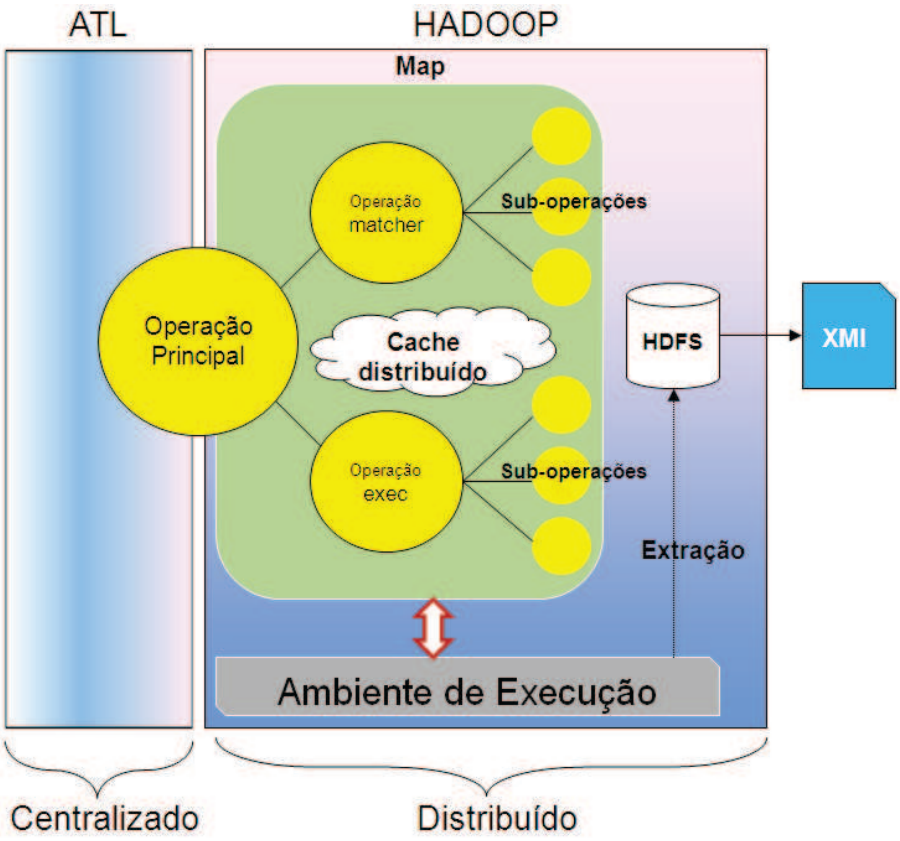

Figura 3.4: Arquitetura de execução ATL com Hadoop baseada em operações.

1.Transição da transformação e Desmembramento do modelo

Estas fases na implementação baseada em operações não sofreram grandes modificações. A transição entre ATL e MapReduce é realizada de forma similar a implementação baseada em regras, porém somente a operação principal é serializada e enviada ao HDFS. O desmembramento do modelo de entrada continua ocorrendo da mesma forma descrita anteriormente, contudo alteramos a estrutura que captura os arquivos para ler bloco menores e assim criar as unidades de trabalho.

2.Mapeamento por operação e execução

A operação principal é deserializada e sua execução iniciada conforme especificado pelo motor ATL. Ao atravessar os níveis, sub-operações são mapeadas criando pares de chave/valor, onde a chave é o nome da sub-operação e o valor é um arquivo binário contendo a instância que representam todo o conteúdo das sub-operações. As tarefas criadas para cada sub-operação mapeada são distribuída para os nós disponíveis no cluster

e assim são executadas as instruções que residem em cada uma delas. O mecanismo de *cache distribuído* do Hadoop é utilizado novamente para criação de instancias do *ambiente de execução*.

Todo esse processo está ilustrado na Figura 3.5, desde a etapa de desmembramento do modelo de entrada, seguida da criação das tarefas de mapeamento e execução dos operações e por fim a transcrição do modelo resultante da transformação para a abordagem baseada em operações.

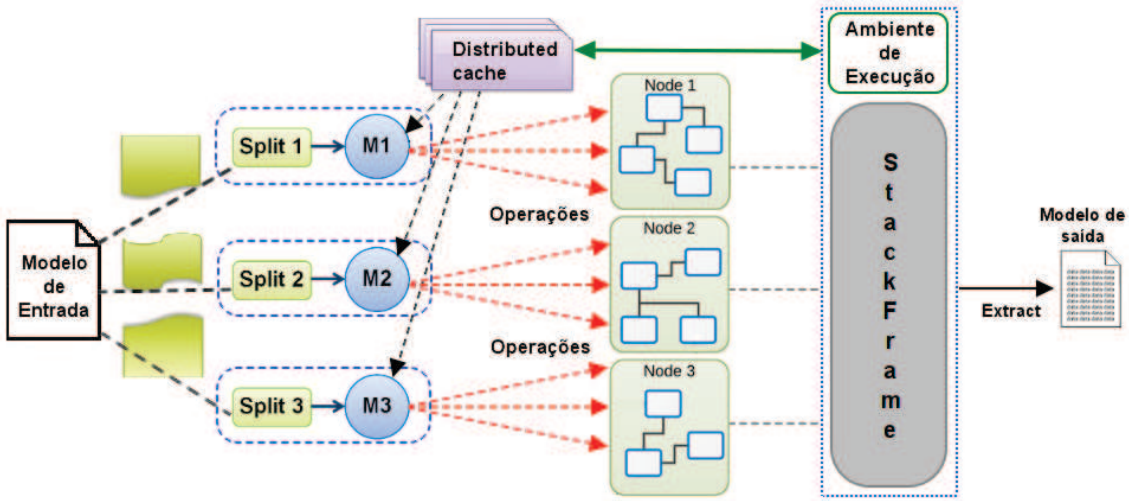

Figura 3.5: Mapeamento e execução da transformação por operação.

O pseudocódigo da listagem 3.4 apresenta o método *map* para a execução da transformação por operações. A partir da operação principal, a hierarquia de operações é decomposta, sendo cada sub-operação distribuída para execução em um nó.

3.Modelo resultante

O modelo resultante da transformação é extraído como um documento XML. O motor de transformação escreve os modelos de saída utilizado o extrator ATL. Um diretório de saída é criado pelo Hadoop e nele é depositado o arquivo resultante da transformação.

```
1  função map(NomeOperacao key, OperacaoATL value)
2  INICIO
3      operacao_map = deserializar(value);
4      Lista lista = operation_map.getInstructions();
5
6      ENQUANTO(existirem Operações na lista)
7         SE(operacao !=null && ambiente_execucao!=null){
8         ASMOperation resultado = Chamar API do ATL (ambiente_execucao, operacao);
9       SENÃO
10        lançar Exceção;
11      FIM SE
12
13       ambiente_execucao = atualizacao(resultado);
14
15       returna (Chave nomeOperacao, Valor resultado da operacao);
16     FIM ENQUANTO
17 FIM
18 FIM
```

Listagem 3.4: Método map.

3.3 Experimentos

Nesta seção apresentamos o pseudocódigo *ClassDiagram2Relational* [35], que transforma diagramas de classe simplificados para esquemas de bancos de dados relacionais. Mostramos como estão representados os metamodelos e modelos de entrada e saída e a forma como as regras de transformação ATL são expressas. Ao final apresentamos algumas medidas de desempenho para então comparar a escalabilidade da abordagem apresentada.

3.3.1 Caso de uso

A transformação de modelos de classe para relacional é um cenário simples, mas completo, que tem sido tradicionalmente usado como estudo de caso para apresentar novas abordagens e linguagens para transformações de modelos.

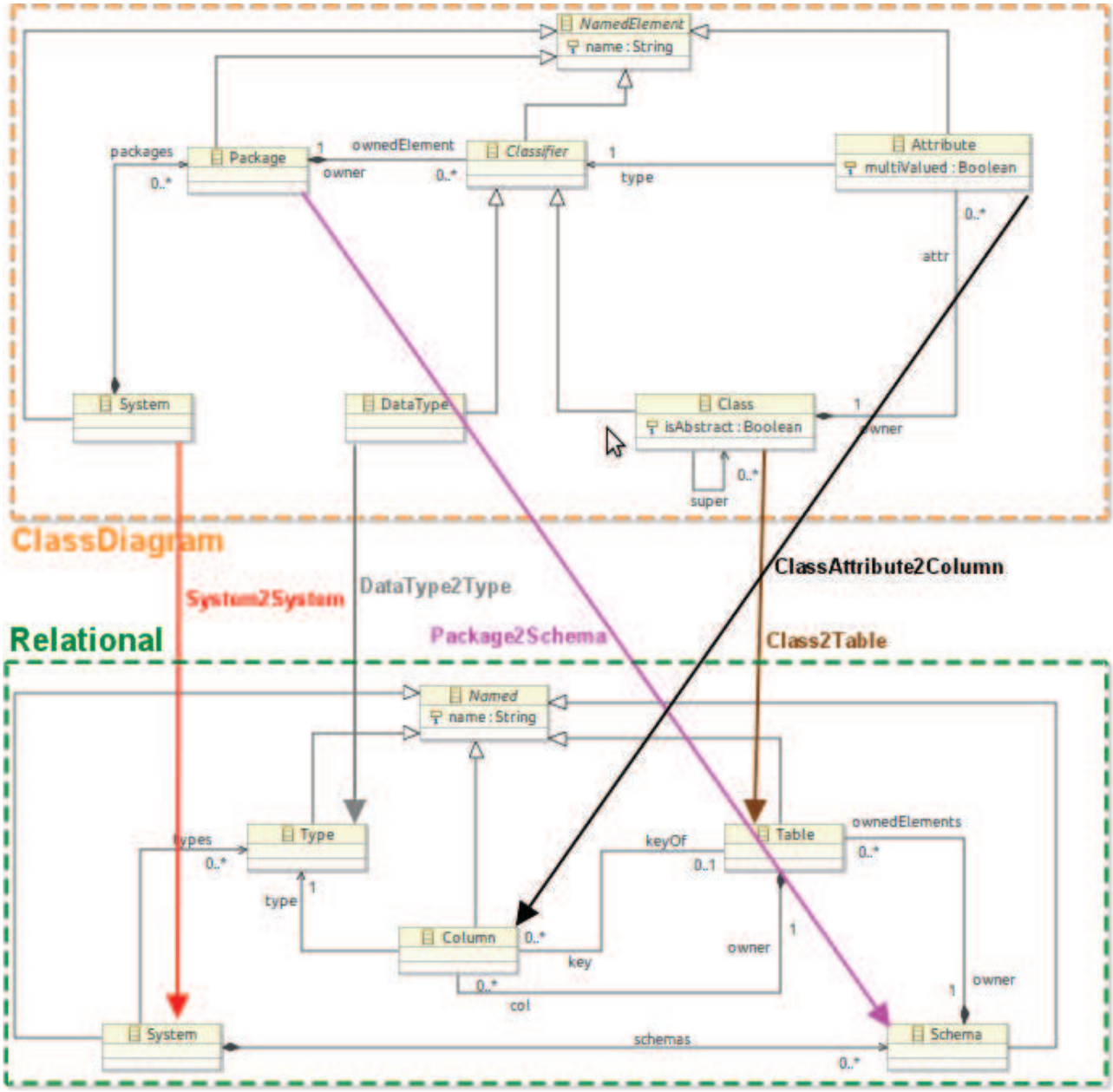

Figura 3.6: Transformação Diagrama de Classe para Relacional.

Esta transformação tem um diagrama de classes simplificado como entrada, já exportado para um arquivo no formato XMI e o transforma em um modelo de banco de dados relacional também num formato de arquivo XMI. Como primeiro passo na execução da transformação, apresentamos na Figura 3.6 acima os metamodelos utilizados na transformação e o conjunto de regras aplicados em cada um dos elementos do metamodelo de entrada para serem produzidos os elementos no metamodelo de saída.

O metamodelo *ClassDiagram* descreve um diagrama de classe, o qual pertence a um

```
module Class2Relational;
create OUT : Relational from IN : ClassDiagram;

helper context String def: firstToLower() : String =
    self.substring(1, 1).toLower() + self.substring(2, self.size());

helper def: objectIdType : Relational!Type =
    ClassDiagram!DataType.allInstances()->select(e | e.name = 'Integer')->first();
```

Listagem 3.5: Cabeçalho ATL para transformação ClassDiagram2Relational.

```
rule System2System{
    from
        p: ClassDiagram!System
    to
        out: Relational!System (
            schemas <- p.packages,
            name <- p.name
        )
}
```

Listagem 3.6: Código ATL para a regra System2System.

sistema e está organizado em pacotes, incluindo classes, atributos e tipos de dados. O metamodelo *Relational* compreende aos elementos básicos de um esquema relacional: tabelas, colunas e tipos; estes pertencentes a um pacote do sistema.

A transformação é descrita por um conjunto de regras de mapeamento, que descreve como um diagrama de classe deve ser transformado em um modelo relacional. Para demonstrar como as regras de transformação são implementadas, apresentamos o pseudo-código das regras de transformação ATL para o experimento *ClassDiagram2Relational* e na sequência discutirmos o papel de cada regra no processo de transformação.

O cabeçalho do código ATL desta transformação *ClassDiagram2Relational* (listagem 3.3), define o metamodelo de entrada (*ClassDiagram*) e o de saída (*Relational*).

Como os diagramas disponíveis no modelo pertencem a um sistema, assumimos uma transformação direta da representação Sistema UML para Sistema ER através da regra *System2System* (listagem 3.4).

A regra *Package2Schema* (listagem 3.5) define que o pacote raiz de cada modelo de classe deve ser mapeado para um esquema com o mesmo nome do pacote.

```
rule Package2Schema{
    from
        p: ClassDiagram!Package
    to
        out: Relational!Schema (
            ownedElements <- p.ownedElement->
           select(e | e.oclIsTypeOf(ClassDiagram!Class)),
            name <- p.name
        )
}
```

Listagem 3.7: Código ATL para a regra Package2Schema.

```
rule Class2Table {
    from
        c : ClassDiagram!Class
    to
        out : Relational!Table (
            name <- c.name,
            col <- c.attr->select(e | not e.multiValued)
        )
}
```

Listagem 3.8: Código ATL para a regra Class2Table.

Cada classe do diagrama deve ser mapeada para uma tabela com o mesmo nome que a classe, conforme especifica a regra *Class2Table* (listagem 3.6).

A regra *DataType2Type* (listagem 3.7) diz que, para cada elemento de Tipo de dados, um elemento de tipo tem que ser criado e seus nomes têm de corresponder.

Determina a regra *DataTypeAttribute2Column* (listagem 3.8) que o tipo de dados do atributo da classe deve ser usados para definir completamente o tipo de um atributo da entidade com seu tipo correspondente em uma tabela de banco de dados.

Os atributos da classe tem que ser devidamente mapeadas para colunas (*ClassAttri-*

```
rule DataType2Type {
    from
        dt : ClassDiagram!DataType
    to
        out : Relational!Type (
            name <- dt.name
        )
}
```

Listagem 3.9: Código ATL para a regra DataType2Type.

```
rule DataTypeAttribute2Column {
    from
        a : ClassDiagram!Attribute (
            a.type.oclIsKindOf(ClassDiagram!DataType) and not a.multiValued
        )
    to
        out : Relational!Column (
            name <- a.name,
            type <- a.type
        )
}
```

Listagem 3.10: Código ATL para a regra DataTypeAttribute2Column.

```
rule ClassAttribute2Column {
    from
        a : ClassDiagram!Attribute (
            a.type.oclIsKindOf(ClassDiagram!Class) and
            not a.multiValued
        )
    to
        foreignKey : Relational!Column (
            name <- a.name
        )
}
```

Listagem 3.11: Código ATL para a regra ClassAttribute2Column.

bute2Column), e algumas colunas podem ser relacionada com outras tabelas, definição de chave estrangeira. Para o caso de atributos com múltiplos valores um tipo especial de mapeamento é realizado (listagem 3.9).

A Figura 3.7 apresenta um exemplo de transformação aplicando alguma das regras do caso de uso selecionado em nosso experimento. A Regra *Package2Schema* transforma um pacote denominado RH em um esquema relacional com o mesmo nome, em seguida a regra *Class2Table* é aplicada convertendo a classe Pessoa em uma tabela de mesmo nome. Por fim, cada atributo de classe gera uma coluna na respectiva tabela através da regra *ClassAttribute2Column*.

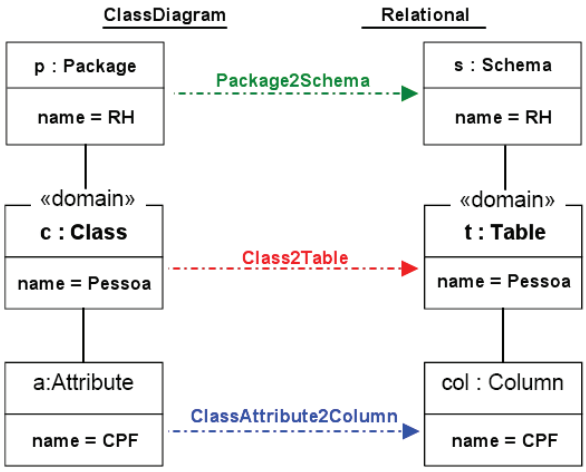

Figura 3.7: Exemplo execução das regras para o experimento Class2Relational.

3.3.2 Implementação e Resultados

Nesta seção, avaliamos o desempenho do mecanismo de transformação de modelo executando em nuvem, comparando com outras duas diferentes variantes de implementação usando ATL, aplicadas sobre o mesmo cenário de transformação e considerando o desempenho em tempo de execução.

Os testes foram realizados em um ambiente com as seguintes características: Processador Intel(R) Core(TM) i5-2400 CPU @ 3.20GHz x 4; com 4Gb de Memória RAM, Sistema Operacional Linux Ubuntu versão 12.04 (x86 64 bits), com kernel 3.2.0-40-generic GNU Linux; Apache Hadoop 1.0.4, usando java versão 1.8.0-ea.

O principal objetivo da abordagem apresentada é a de proporcionar escalabilidade, isto é, transformar grandes modelos. Assim, para validar a nossa abordagem, pretendemos verificar o tempo absoluto de transformação para um conjunto representativo de grandes modelos de entrada e a relativa melhoria no que diz respeito às abordagens existentes.

Um componente essencial para validar nossa implementação é o uso de grandes modelos consistentes para transformação, contudo é extremamente difícil encontrar modelos com essas características disponíveis. Em nosso experimento optamos por utilizar um grande modelo de classe composto por dezenas de pacotes, centenas de classes em cada

pacote e milhares de atributos para cada classe. Totalizando aproximadamente 100Mb usamos este como base para construir e gerar novos e maiores modelos através da replicação. Trabalhos como ATL Paralelo [56] utilizam a técnica de dividir os modelo em vários arquivos e diretórios, para assim evitar grandes modelos e possíveis falhas de referências inter-modelos. Há outras soluções como [27], que apresentam algoritmos eficientes para sincronização e divisão de grandes modelos, e que garantem a consistência da transformação.

O pontos de entrada para a distribuição da execução é a serialização das operações, que então são enviadas aos nós escravos do *cluster* para processamento. O ambiente de execução criado pelo ATL é fortemente dependente de estruturas e interfaces EMF, não sendo passível de serialização direta através de comandos Java padrão. Assim, optamos por recriar o ambiente de execução e compartilha-lo através do mecanismo de *cache distribuído* juntamento com os metamodelos de entrada e saída, ficando acessível para todas as máquinas do *cluster*.

Na medição é considerado o tempo total do processo de transformação, isso inclui o tempo de carregamento, serialização e escrita dos modelos. Usamos um conjuntos de diferentes tamanhos de modelos, iniciando com um de 100MB composto por mais de 500 mil elementos até um modelo de 8GB este formado por mais de 70 milhões de elementos.

O ambiente de execução dos experimentos para as implementações distribuídas é formado por um *cluster* de 5 máquinas com a mesma configuração citada acima. Já a versão padrão de ATL e a adaptação para execução em paralelo seguem de forma centralizada.

A tabela 3.1 apresenta os resultados em segundos do desempenho para os dois mecanismos de transformação de modelo executados em nuvem, da solução de transformação tradicional ATL e a abordagem baseada na execução paralela.

A tabela 3.2 apresenta o percentual sobre o tempo médio despendido em cada uma das etapas do processo de transformação, ou seja, no carregamento do modelo, na execução da transformação e na escrita do modelo resultante.

	TRADICIONAL	PARALELO	DISTRIBUÍDO/ REGRAS	DISTRIBUÍDO/ OPERAÇÕES
100MB	1014.155	92.064	243.183	190.752
400MB	2790.911	335.005	1326.512	351.425
800MB	5991.875	770.689	2075.108	504.831
1GB	-	918.855	2545.190	595.190
4GB	-	-	5569.547	2739.979
8GB	-	-	11167.624	4213.419

Tabela 3.1: Tempo de execução das transformações em segundos.

CARREGAMENTO	TRANSFORMAÇÃO	ESCRITA
20%	77%	3%

Tabela 3.2: Porcentagem do tempo médio em cada etapa do processo de transformação.

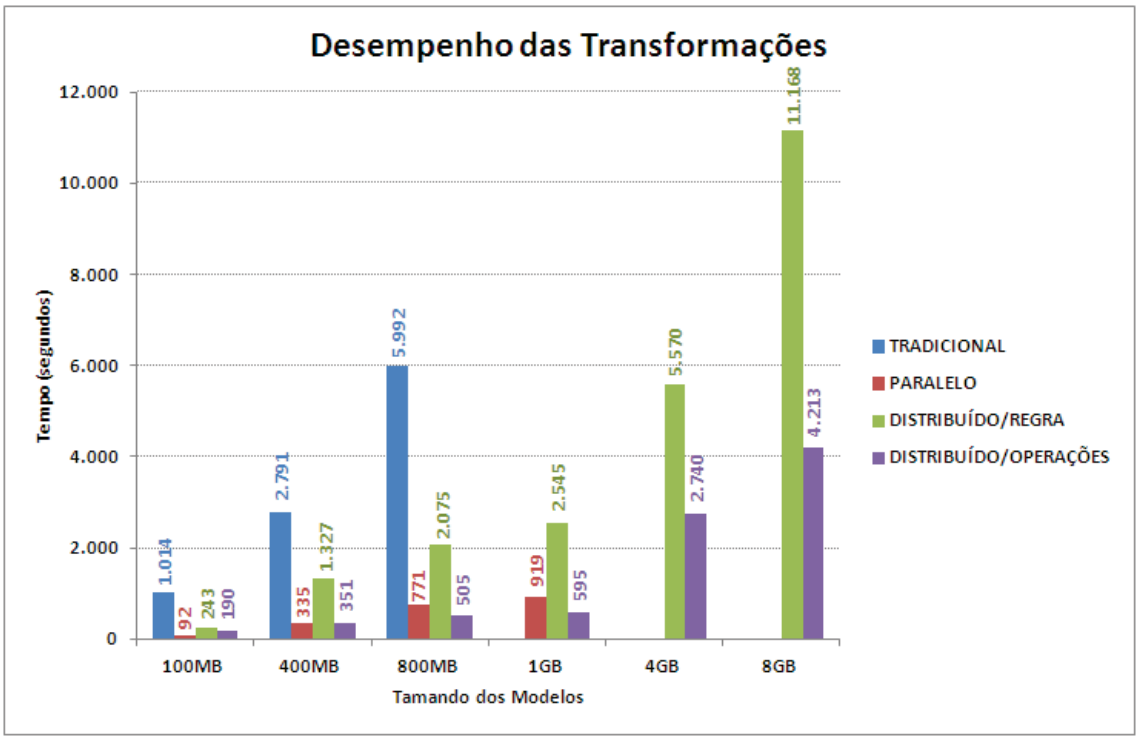

Figura 3.8: Gráfico com o tempo de execução das abordagens.

Com base nos resultados expostos na Tabela 3.1 e no gráfico (Figura 3.8) fizemos as seguintes observações:

Para modelos com menos de 100MB, a solução distribuída é nitidamente mais lenta, atribuísse a isso o tempo de serializar as operações e passar ao Hadoop. Conforme o tamanho dos modelos aumenta, a solução de execução paralela apresenta um desempenho superior até modelos de 500MB. Contudo a partir desse ponto a solução distribuída passa a ser mais eficiente, por conseguir trabalhar com grandes modelos.

Comparando as soluções distribuídas, mesmo adotando arquiteturas semelhantes, vemos um enorme contraste de performance entre a implementação baseada em regras com a baseada em operações. Fato esse que devesse fortemente ao conjunto de regras envolvidas na transformação. Como no caso de uso utilizado para os testes inserimos um total de 5 regras isso gera uma limitação para abertura de mapeadores e um gargalo posteriormente nos redutores baseados no nome das regras. Já na implementação organizada por operações o cluster é melhor ocupado, uma vez que conjunto de operações envolvidas na transformação é proporcionalmente maior que o conjunto de regras.

Uma técnica a ser utilizada caso um grande conjunto de regras seja exigido no processo de transformação é ampliar o número de máquinas do *cluster* e assim distribuir e cobrir mais regras por nó.

O tempo de execução de uma transformação com Hadoop é fortemente afetada pelo volume de dados transferidos e número de fases envolvidas. A comunicação e geração de saídas intermediárias entre mapeador e redutor gera um custo de processamento, especialmente em arquivos serializados onde dependendo do tamanho real, os segmentos são colocados no *buffer* ou escrito para o disco. O melhor desempenho foi obtido usando apenas uma fase e modelos menores como entrada para a fase de mapeamento.

Avaliamos duas diferentes alternativas de atingir a escalabilidade, através da distribuição por regras e operações. A escolha por estes componentes se deve ao fato de serem estruturas de mais alto nível dentro do ATL, que possuem características passíveis de agrupamento, organização e divisão. Uma outra forma mais especializada, que poderia ser implantada, é a distribuição com base nas entidades do modelo, ou seja, elementos do modelo e suas relações.

Para controlar a navegação, o motor ATL necessita do carregamento completo do modelo de entrada em memória, sendo este o grande empecilho para executar transformação de grandes modelos de forma centralizada. A etapa de carregamento em nossos testes consumiu em média 20% do tempo no processo de transformação. A estratégia utilizada no MapReduce para dividir o modelo de entrada em fragmentos nas abordagens baseada

em regras (200Mb) e operações (100Mb) possibilitou a execução na solução distribuída.

O uso da técnica de replicação para gerar os modelos utilizados nos testes, cria um cenário que amplia o número de elementos do modelo base, porém não insere diferentes elementos de modelagem ou conexões permitindo assim capturar fragmentos independentes. Grandes modelos onde há profundidade em interações entre os componentes, representariam uma fonte de erro para nossa implementação, o que demandaria outra forma de particionar o modelo de entrada.

Os experimentos foram realizados em um cluster formado por cinco máquinas, a ampliação desse número resultaria em um melhor desempenho da solução baseada em operações, contudo para a implementação baseada em regras não surtiria efeitos expressivos, pois o conjunto de regras envolvidos na transformação corresponde ao número nós que serão utilizados na distribuição.

Regras que não exigem uma ordem explícita podem ser executadas em paralelo, sem nenhum conflito ou inconsistência no modelo resultante. Interdependências entre modelos e regras estão fora do escopo deste trabalho.

CAPÍTULO 4

CONCLUSÃO

Neste trabalho apresentamos uma solução para execução de transformações distribuídas, baseada no framework MapReduce. Implementamos duas abordagens e comparamos os resultados com outras duas soluções. O objetivo principal foi ter uma solução escalável, uma baseada em regras e outra em operações. A primeira solução, dissolvida em duas fases, demonstra capacidade de trabalhar com grandes modelos e distribuir a transformação, porém o tempo total de execução é alto e a solução condicionada ao número de regras aplicadas na transformação. Quanto maior o conjunto de regras envolvido no processo, maior as possibilidades de distribuição. A segunda implementação apresentou resultados 60% melhores, a redução de fases envolvidas no processo e a divisão do modelo em segmentos menores deram maior agilidade a solução baseada em operações.

Um comparativo entre as abordagens centralizada e distribuída é apresentado para análise e reflexão sobre as implementações. Principalmente no que tange transformações de grades modelos. Constatamos que a abordagem implementada é eficiente, levando em conta as características do ambiente de execução, a dimensão dos modelos e o custo associado às operações de leitura/transformação/escrita sobre os modelos, pode-se observar que os tempos de execução são elevados, contudo a ferramenta atende o propósito de distribuir transformações de modelos. Ainda assim, é observável que o tamanho e o número elevado de elementos nos modelos introduz um peso cada vez maior sobre o tempo total de execução do programa, pois o ATL necessita do carregamento total dos modelos para processar a transformação.

O cenário ideal para a arquitetura especificada nesse experimento seria passar ao Hadoop apenas grandes objetos serializados, um contendo as operações custosas (*matcher e exec*) e outro contendo o estado atual do ambiente de execução associado a cada uma das

operações. Entretanto, atualmente modelos construídos com base no framework EMF não são diretamente serializados através de comandos Java padrão. O ambiente de execução criado pelo ATL é fortemente dependente de estruturas e interfaces EMF. Assim não foi possível utilizar somente serialização.

Implementações atuais do ATL exigem o carregamento completo do modelo em memória. Para grande modelos a fase de carga consome cerca de 20% do tempo no processo de transformação, 77% para a execução e 3% para escrita do modelo resultante.

Se considerarmos os mecanismos de execução centralizada, seguindo uma abordagem de aplicação sequencial ou paralela, estas esbarram no problema de alocação de memória. Nossa abordagem demonstrou ser mais rápida para caso particular de grandes modelos, e também a capacidade de expansão. No entanto, este é apenas o começo, há várias linhas que ainda temos para explorar em trabalhos futuros.

4.1 Trabalhos futuros

ATL é projetado para suportar cenários de transformação onde os modelos de origem e destino são artefatos criados com várias tecnologias, tais como bancos de dados, documentos XML, etc. Dessa forma ATL serve o propósito. No entanto, se considerarmos grandes modelos, o carregamento se torna um gargalo no ATL. Com base nas tecnologias relacionadas e funções chaves necessárias para implementar uma motor de transformação na nuvem, como investigações e atividades futuras podemos implementar uma iteração com abordagens existentes para armazenamento de grandes modelos.

Outro caminho a ser estudado é a possibilidade de estender a funcionalidade da implementação para utilizar formas especializadas de divisão das operações, como o uso da estrutura EMF para distribuição da transformações com base nas entidades, elementos e relações do modelo. Esta implementação estaria diretamente integrada com o processo de divisão do modelo e com a interdependência dos elementos.

BIBLIOGRAFIA

[1] AtlanMod. Atlantic modeling. http://www.emn.fr/z-info/atlanmod/index.php, setembro de 2013.

[2] Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk, e Gabor Karsai. The graph rewriting and transformation language: Great. *ECEASST*, 1, 2006.

[3] K. Barmpis e D.S. Kolovos. Comparative Analysis of Data Persistence Technologies for Large Scale Models. *15th International Conference on Model Driven Engineering Languages e Systems*, 2012.

[4] Matthias Biehl. Literature study on the state of the art in model transformation technology. Relatório Técnico 2010:07, KTH, Mechatronics, 2010. QC 20101209.

[5] C. Brun e A. Pierantonio. Model Differences in the Eclipse Modelling Framework. *The European Journal for the Informatics Professional IX*, abril de 2008.

[6] Hugo Brunelière, Jordi Cabot, e Frédéric Jouault. Combining Model-Driven Engineering and Cloud Computing. *Modeling, Design, and Analysis for the Service Cloud - MDA4ServiceCloud'10: Workshop's 4th edition (co-located with the 6th European Conference on Modelling Foundations and Applications - ECMFA 2010)*, Paris, France, junho de 2010.

[7] Hugo Brunelière, Jordi Cabot, Frédéric Jouault, Massimo Tisi, e Jean Bézivin. Industrialization of Research Tools: the ATL Case. *Third International Workshop on Academic Software Development Tools and Techniques - WASDeTT-3 (co-located with the 25th IEEE/ACM International Conference on Automated Software Engineering - ASE'2010)*, Antwerp, Belgium, setembro de 2010.

[8] Loli Burgueño, Javier Troya, Manuel Wimmer, e Antonio Vallecillo. On the concurrent execution of model transformations with linda. *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, páginas 3:1–3:10, 2013.

[9] R. Buyya, J. Broberg, e A. Goscinski. *Cloud Computing: Principles and Paradigms*. 2011.

[10] J. Castrejon, G. Vargas-Solar, C. Collet, e R. Lozano. Model-Driven Cloud Data Storage. *First International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, 2012.

[11] C. Clasen, M. D. Del Fabro, e M. Tisi. Transforming Very Large Models in the Cloud: a Research Roadmap. *First International Workshop on Model-Driven Engineering on and for the Cloud (ECMFA)*, 2012.

[12] Krzysztof Czarnecki e Simon Helsen. Classification of model transformation approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003.

[13] Varro Daniel e András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234, outubro de 2007.

[14] Jeffrey Dean e Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[15] M. Didonet Del Fabro. *Metadata management using model weaving and model transformation*. Phd thesis, University of Nantes, setembro de 2007.

[16] A. L. Drakopoulos. *Scalable persistence of EMF models*. Msc in information technology, Department of Computer Science, University of York, setembro de 2010.

[17] J. Espinazo e J. G. Molina. A homogeneous repository for collaborative MDE. *Proceedings of the 1st International Workshop on Model Comparison in Practice (IWMCP 10)*, 2010.

[18] Javier Espinazo-Pagán, Jesús Sánchez Cuadrado, e Jesús García Molina. Morsa: A scalable approach for persisting and accessing large models. *MoDELS*, páginas 77–92, 2011.

[19] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, e Patrick Valduriez. Applying generic model management to data mapping. *BDA*, 2005.

[20] Jean Marie Favre. Towards a Basic Theory to Model Driven Engineering. *Workshop on Software Model Engineering (WISME)*, 2004.

[21] F. Fondement e R. Silaghi. Defining Model Driven Engineering Processes. *3rd Workshop in Software Model Engineering (WISME @ UML 2004)*, 2004.

[22] The Apache Software Foundation. Apache hadoop. http://hadoop.apache.org/, setembro de 2013.

[23] The Eclipse Foundation. Atl transformations. http://www.eclipse.org/atl/atlTransformations/, setembro de 2013.

[24] The Eclipse Foundation. Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon). http://www.eclipse.org/gmt/epsilon, setembro de 2013.

[25] Dragan Gasevic, Dragan Djuric, e Vladan Devedzic. *Model Driven Engineering and Ontology Development (2. ed.)*. Springer, 2009.

[26] Dragan Gasevic, Nima Kaviani, e Marek Hatala. On metamodeling in megamodels. Gregor Engels, Bill Opdyke, Douglas C. Schmidt, e Frank Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, páginas 91–105. Springer, 2007.

[27] Holger Giese e Stephan Hildebrandt. Efficient model synchronization of large-scale models. Relatório Técnico 28, Hasso Plattner Institute at the University of Potsdam, 2009.

[28] Jeff Gray, Yuehua Lin, e Jing Zhang. Automating change evolution in model-driven engineering. *IEEE Computer*, 39(2):51–58, 2006.

[29] Object M. Group. OMG unified modeling language (OMG UML). http://www.omg.org/gettingstarted/what_is_uml.htm, novembro de 2013.

[30] Object Management Group. Meta Object Facility (MOF) Query/View/Transformation Transformation Specification. www.omg.org/spec/QVT/, setembro de 2013.

[31] Bézivin J., F. Jouault, e Touzet D. An Introduction to the ATLAS Model Management Architecture. Relatório técnico, LINA, 2005.

[32] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, e Ivan Kurtev. Atl: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[33] Frédéric Jouault e Ivan Kurtev. Transforming models with atl. *MoDELS Satellite Events*, páginas 128–138, 2005.

[34] Frédéric Jouault e Jean-Sébastien Sottet. An AmmA/ATL Solution for the GraBaTs 2009 Reverse Engineering Case Study. *Fifth International Workshop on Graph-Based Tools - Grabats 2009 (co-located with TOOLS 2009)*, Zurich, Suisse, julho de 2009.

[35] Frédéric Jouault e Massimo Tisi. Towards incremental execution of atl transformations. Laurence Tratt e Martin Gogolla, editors, *ICMT*, volume 6142 of *Lecture Notes in Computer Science*, páginas 123–137. Springer, 2010.

[36] Stuart Kent. Model driven engineering. *IFM*, páginas 286–298, 2002.

[37] Mathias Kleiner, Marcos Didonet Del Fabro, e Davi De Queiroz Santos. Transformation as search. Pieter Van Gorp, Tom Ritter, e Louis M. Rose, editors, *ECMFA*, volume 7949 of *Lecture Notes in Computer Science*, páginas 54–69. Springer, 2013.

[38] Dimitrios S. Kolovos, Richard F. Paige, e Fiona A. Polack. The epsilon transformation language. *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, ICMT '08, páginas 46–60, Berlin, Heidelberg, 2008. Springer-Verlag.

[39] Dimitris Kolovos, Richard F. Paige, e Fiona A. C. Polack. The grand challenge of scalability for model driven engineering. Michel R. V. Chaudron, editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*, volume 5421 of *Lecture Notes in Computer Science*, páginas 48–53. Springer, 2008.

[40] D.S. Kolovos, R.F. Paige, e F.A. Polack. Scalability: The holy grail of model driven engineering. *ChaMDE 2008 Workshop Proceedings: International Workshop on Challenges in Model-Driven Software Engineering*, páginas 10–14, 2008.

[41] Ivan Kurtev. Applications of graph transformations with industrial relevance. capítulo State of the Art of QVT: A Model Transformation Language Standard, páginas 377–393. Springer-Verlag, Berlin, Heidelberg, 2008.

[42] Jimmy Lin e Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010.

[43] Dan C. Marinescu. Cloud Computing and Computer Clouds. Relatório técnico, Department of Electrical Engineering e Computer Science, University of Central Florida, 2012.

[44] Tom Mens e Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, março de 2006.

[45] Naouel Moha, Sagar Sen, Cyril Faucher, Olivier Barais, e Jean-Marc Jézéquel. Evaluation of kermeta for solving graph-based problems. *STTT*, 12(3-4):273–285, 2010.

[46] Neo4j. Neo4j - the world leading graph database. http://neo4j.org/, novembro de 2013. November 22, 2013.

[47] Dube M. R. e S. K. Dixit. Modeling theories and model transformation scenario for complex system development. *International Journal of Computer Applications*, 38:11–18, março de 2012.

[48] M. Scheidgen e A. Zubow. Map/Reduce on EMF-Models. *International Workshop on Model-Driven Engineering for High Performance and Cloud computing (MODELS)*. ACM Digital Library, 2012.

[49] Markus Scheidgen, Anatolij Zubow, Joachim Fischer, e Thomas H. Kolbe. Automated and transparent model fragmentation for persisting large models. *Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems*, MODELS'12, páginas 102–118, Berlin, Heidelberg, 2012. Springer-Verlag.

[50] K. Shvachko, H. Kuang, S. Radia, e R. Chansler. The Hadoop Distributed File System. *Mass Storage Systems and Technologies (MSST)*, páginas 1–10. IEEE Computer Society, 2010.

[51] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, e Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2 edition, 2009.

[52] Ragnhild Van Der Straeten, Tom Mens, e Stefan Van Baelen. Challenges in model-driven software engineering. *MoDELS Workshops*, páginas 35–47, 2008.

[53] Eugene Syriani e Hans Vangheluwe. Matters of model transformation. Relatório Técnico SOCS-TR-2009.2, 2009.

[54] Orient Technologies. Orienteddb. http://www.orientechnologies.com/, novembro de 2013.

[55] Peter Thiemann. Programmable type systems for domain specific languages. *Electr. Notes Theor. Comput. Sci.*, 76:233–251, 2002.

[56] Massimo Tisi, Salvador Martinez, e Hassene Choura. Parallel Execution of ATL Transformation Rules. *MoDELS*, páginas 656–672, Miami, États-Unis, 2013.

[57] Massimo Tisi, Salvador Martínez Perez, Frédéric Jouault, e Jordi Cabot. Lazy execution of model-to-model transformations. *MoDELS*, páginas 32–46, 2011.

[58] Jing Zhang, Gongqing Wu, Xuegang Hu, e Xindong Wu. A distributed cache for hadoop distributed file system in real-time cloud services. *Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing*, GRID '12, páginas 12–21, Washington, DC, USA, 2012. IEEE Computer Society.