UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA ABORDAGEM PARA CRIAÇÃO DE MÁQUINAS DE TRANSFORMAÇÕES DE KDM PARA PSM

GUISELLA CLARA ANGULO ARMIJO

ORIENTADOR: PROF. DR.VALTER VIERA DE CAMARGO

São Carlos – SP

Novembro/2018

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA ABORDAGEM PARA CRIAÇÃO DE MÁQUINAS DE TRANSFORMAÇÕES DE KDM PARA PSM

GUISELLA CLARA ANGULO ARMIJO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software
Orientador: Prof. Dr.Valter Viera De Camargo

São Carlos – SP

Novembro/2018

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado da candidata Guisella Clara Angulo Armijo, realizada em 06/12/2018:

Prof. Dr. Valter Vieira de Camargo
UFSCar

Prof. Dr. Daniel Lucrédio
UFSCar

Prof. Dr. Marcio Eduardo Delamaro
USP

Dedico especialmente este trabalho a meus pais.
Mãe e Pai, amo vocês.

AGRADECIMENTOS

A Deus por permitir meu ingresso no mestrado e pela força para nunca desistir.

Ao meu orientador, Prof. Dr. Valter Vieira de Camargo, pelas constantes lições no decorrer do mestrado, pela paciência e pelo incentivo e guia para desenvolver um trabalho científico de qualidade.

Aos meus pais, Guillermo e Teofila, pelo infinito amor, pelas constantes ligações que ajudaram a aquecer meu coração nos dias frios de saudade, pelas palavras de incentivo, preocupação, força e sobretudo pela confiança ao aceitar e apoiar minhas decisões, apesar que isso significasse ficar longe de vocês. Os senhores forneceram-me sempre o melhor exemplo de perseverança, luta e coragem. Eu sou eternamente grata por me tornarem quem eu sou e quero dizer-lhes que essa conquista é nossa.

Aos meus irmãos Orlando, Maritza, July, Carlos e Christian (a quem eu considero meu irmãozinho) pelo amor incondicional, por acreditar nas minhas competências e pelas inumeráveis palavras de incentivo que sempre me motivaram a continuar.

Aos meus sobrinhos belos, pelas mostras de amor, alegria, brincadeiras e por sempre encher meu coração com detalhes e palavras sinceras que só uma criança pode dizer. Quero agradecer a minha sobrinha Ada, quem sendo ainda uma criança me deu uma lição de fé e luta.

A meu cunhado Freddy, minhas tias e primos, obrigada pelo carinho e por compartilhar cada avanço na consecução das minhas metas.

Ao meu namorado Sandro, pelo companheirismo, pelas constantes palavras de ânimos, por sempre me ouvir, por renunciar a sair muitos finais de semana para que eu conseguisse terminar meus trabalhos, por assistir televisão sem som para não me atrapalhar e pelo imenso apoio e amor que me demostra cada dia.

À família de Sandro: Paula, Angélica, Edilaine, Caio, Rafael e senhora Joana. Obrigada pelo carinho e por me fazer sentir sempre parte da sua linda família.

Às minhas amigas e irmãs *'del alma'*, Candy e Lizbeth, pela amizade sincera que começou há 15 anos, por me ouvirem quando precisei, pelas inumeráveis risadas e preocupação no meu bem estar. Candy, obrigada por ser a representante da minha família no Brasil, ver você me faz sentir em casa.

A todos meus amigos de Peru especialmente a Patty, Julissa e Juan. Obrigada pelo carinho, pela amizade, por me procurar em cada oportunidade que estive no Peru e por me encorajar sempre a seguir e cumprir meus sonhos.

A todos meus companheiros de laboratório: Daniel, Bruno, André, Zannan, Giannandrea que sempre me ajudaram a responder as minhas dúvidas e não hesitaram em me oferecer apoio.

A meus amigos Steve e Marco, pela amizade, pelas palavras de ânimo nesse caminho chamado *Mestrado* que começamos juntos.

A todos que, de alguma forma, contribuíram com a realização desse trabalho. Obrigada!

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior-Brasil (CAPES) - Código de Financiamento 001.

Ora, a fé é o firme fundamento das coisas que se esperam, e a prova das coisas que não se veem.

Hebreus 11:1

RESUMO

As necessidades de negócio obrigam as empresas a modernizar seus sistemas, porém as mudanças em sistemas legados são complexas e custosas em consequência de que o código legado possui lógica de programação, decisões de projeto, requisitos de usuário e regras de negócio que são difíceis de extrair. Nesse contexto, o Object Management Group (OMG) criou a Modernização Dirigida à Arquitetura (Architecture-Driven Modernization (ADM)), tornando possível modelar todos os artefatos do sistema legado como modelos e estabelecendo transformações entre os diferentes níveis de abstração. A ADM é um tipo de reengenharia de software que emprega modelos padrão ao longo do processo e lida com toda a arquitetura do sistema. O metamodelo principal é o Metamodelo de Descoberta de Conhecimento (Knowledge Discovery Metamodel (KDM)), que é um modelo independente de linguagem e plataforma capaz de representar diversos aspectos de um sistema de software. Embora um número significativo de pesquisas pode ser encontrado na fase de engenharia reversa da ADM, pouco pode ser encontrado com relação à engenharia avante; principalmente na geração de modelos específicos de plataforma (Platform Specific Model (PSM)) a partir do KDM. Esta fase é essencial, pois pertence à parte final do ciclo da ferradura ADM, completando todo o processo de reengenharia automatizada. No entanto, a falta de pesquisa e a ausência de suporte de ferramentas disponíveis dificultam a adoção da ADM na prática. A fim de contribuir para a fase de engenharia avante do ADM, neste projeto foi desenvolvida uma ferramenta chamada *RUTE-K2J*, que é um motor de transformação para gerar um modelo Java a partir de um modelo KDM. Além disso, a partir dessa experiência prática, foi generalizado um processo para dar suporte a engenheiros de modernização tanto na i) criação de mecanismos de transformação do KDM para qualquer outro PSM quanto ii) na evolução da RUTE-K2J em direção a um mecanismo de transformação mais estável e completo. A ferramenta *RUTE-K2J* foi avaliada com uma estratégia de teste que considerou cenários típicos de software com o intuito de validar a corretude das regras de transformação que compõem o motor.

Palavras-chave: ADM, Modelo KDM, ATL, Modelo Java, transformação de modelos

ABSTRACT

Business needs compel companies to modernize their systems, but changes in legacy systems are complex and costly because the legacy code has programming logic, design decisions, user requirements, and business rules that are difficult to extract. In this context, gls OMG created Architectural Modernization (ADM), making it possible to model all the artifacts of the legacy system as models and establishing transformations between the different levels of abstraction. ADM is a type of software reengineering that employs standard models throughout the process and handles the entire system architecture. The main metamodel is the Knowledge Discovery Metamodel (KDM), which is an independent language and platform model capable of representing various aspects of a software system. Although a significant number of researches can be found in the reverse engineering phase of ADM, little can be found in relation to the forward engineering; especially in the generation of platform-specific models (PSM) from the KDM. This phase is essential because it belongs to the final part of the ADM horseshoe cycle, completing the entire automated reengineering process. However, the lack of research and the lack of support of available tools hinder ADM adoption in practice. In order to contribute to the advanced engineering phase of ADM, in this project a tool called *RUTE-K2J* was developed, which is a transformation engine to generate a Java Model from a KDM model. In addition, from this practical experience, a process was developed to support modernization engineers both in (i) creating KDM transformation mechanisms for any other PSM and (ii) in the evolution of RUTE-K2J towards a more stable transformation mechanism and complete. The tool *RUTE-K2J* was evaluated with a test strategy that considered typical software scenarios in order to validate the correctness of the transformation rules that make up the engine.

Keywords: ADM, KDM, ATL, Java Model, model transformation

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

*

Sumário

Capítulo 1
INTRODUÇÃO

1.1 Contexto

Sistemas de software são considerados legados quando seus custos de manutenção encontram-se elevados, isto é, consume muito tempo e esforço atender pequenas demandas de manutenção. Outro fator que caracteriza um sistema como legado é o fato de que, apesar dos altos custos de manutenção, ainda são essenciais para apoiar os processos de negócio de suas organizações.

Há décadas, Lehman (1980) argumentou que o software deve mudar continuamente ou se tornará cada vez menos útil no mundo real. Além disso, a segunda lei do Lehman observa que a estrutura do software em evolução se degradará, a menos que medidas corretivas sejam tomadas regularmente. Para a grande maioria dos softwares legados, tais ações corretivas nunca foram tomadas, portanto qualquer estrutura originalmente existente se degradou. Sem a documentação, a manutenção é feita usando o código-fonte porque é a única fonte confiável de informações sobre o sistema. Com o tempo, o software se torna muito difícil de manter, mas os pedidos de manutenção da organização se tornam mais frequentes e mais insistentes (BENNETT, 1995).

A Reengenharia é a inspeção e a alteração de um sistema para reconstituí-lo em uma nova forma e a subsequente implementação da nova forma. A reengenharia geralmente inclui alguma forma de engenharia reversa (para obter uma descrição mais abstrata) seguida por alguma forma de engenharia avançada ou reestruturação (CHIKOFSKY; CROSS, 1990). Dentro da área de engenharia de software, quando se trata do tema reengenharia, geralmente a preocupação é em automatizar partes do processo, para que não sejam feitos manualmente.

Um dos maiores problemas enfrentados durante o processo de criação de ferramentas que automatizam os processos tradicionais da reengenharia é a falta de formalização e padronização no processo de desenvolvimento dessas ferramentas, portanto, diferentes ferramentas que abordam tarefas específicas no processo de reengenharia dificilmente podem interagir. Com o objetivo de promover o consenso sobre a modernização de sistemas existentes a OMG criou a Modernização Dirigida a Arquitetura (ADM), que defende a realização de processos de reengenharia seguindo o padrão MDA (OMG, 2001), tornando-se possível modelar todos os artefatos de software legado como modelos e estabelecendo transformações de modelos entre os

diferentes níveis de abstração. Os princípios orientados por modelos auxiliam na solução dos problemas de padronização e automação inerentes aos processos de reengenharia tradicional.

O modelo de ferradura da ADM foi adaptado do modelo de ferradura da reengenharia tradicional. Na Figura 1.1 é mostrada a ferradura de modernização e as etapas que a compõem: (i) *Engenharia Reversa* identifica os componentes do sistema, seus interrelacionamentos e constrói uma ou mais representações abstratas do sistema legado; (ii) *Reestruturação* transforma a representação abstrata em outra representação do sistema no mesmo nível de abstração, melhorando o sistema legado origem, mas preservando o comportamento externo do sistema; e (iii) *Engenharia Avante* responsável por transformar os artefatos que representam o sistema em nível de modelo para o nível de código-fonte.

Além disso, o modelo de modernização em ferradura usa a nomenclatura MDA para se referir a diferentes níveis de abstração: (i) O modelo independente de computação (Computation Independent Model (CIM)) é uma visão de negócio do sistema a partir do ponto de vista independente de computação em um nível de abstração elevado; (ii) O Modelo Independente de Plataforma (Platform Independent Model (PIM)) é uma visão do sistema a partir do ponto de vista independente da plataforma em um nível de abstração intermediário, assim, os modelos PIM abstraem todos os detalhes de implementação relacionados à plataforma; e finalmente (iii) O Modelo Específico da Plataforma (PSM) é uma visão tecnológica de um sistema do ponto de vista da plataforma em um baixo nível de abstração (PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2010).

Observa-se, na Figura 1.1, na etapa de Engenharia reversa, os modelos CIM, PIM e PSM são gerados por meio de uma série de transformações que aumentam o nível de abstração e que iniciam a partir do sistema legado. Na etapa de Reestruturação as atividades de refatoração e otimização podem ser estabelecidos em qualquer dos níveis de abstração CIM, PIM ou PSM. Por fim, na etapa de Engenharia avante, a instância do modelo refatorado/otimizado é transformado em código fonte por meio de uma série de transformações que diminuem o nível de abstração em cada transformação. Dessa forma, para que o processo de modernização ADM possa ser concluído é necessária a utilização de transformações automáticas entre os diferentes níveis de modelos no intuito de chegar ao nível de código-fonte outra vez.

A ADM também tem liderado o desenvolvimento de um conjunto de padrões para abordar os diferentes desafios que surgem na modernização dos sistemas legados. O principal padrão é o Metamodelo de Descoberta de Conhecimento (Knowledge Discovery Metamodel - KDM) que permite representar todos os artefatos de software envolvidos em um determinado sistema legado de forma integrada e padronizada (PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2010).

O KDM fornece um formato de troca de informações comum entre modelos que facilita a interoperabilidade entre ferramentas de modernização. Além disso, permite a representação de artefatos físicos e lógicos de sistemas de software existentes, bem como seus relacionamentos em vários níveis de abstração (OMG, 2016).

Na literatura existem trabalhos de reengenharia que descrevem como transformar o mo-

delo KDM em outro modelos. Por exemplo, para o Modelo de Banco de Dados (PÉREZ-CASTILLO et al., 2009), Modelo de Métricas (CANOVAS; MOLINA, 2010) e Modelo de Diagrama de Sequência UML (MARTINEZ; PEREIRA; FAVRE, 2014). Porém, poucos trabalhos concluem o processo de modernização usando o modelo KDM no estágio de engenharia avante, por exemplo, a transformação para o Modelo SBVR (PÉREZ-CASTILLO et al., 2013) e Modelo de Aplicações Cliente RIA (RODRÍGUEZ-ECHEVERRÍA et al., 2011). Nesse contexto, o processo de Modernização ADM precisa de motores de transformação que, considerando o modelo KDM, forneçam suporte no estágio de engenharia avante. Isto permitirá acrescentar os suportes ferramentais para casos de uso de modernização ADM, abrir a possibilidade para que os projetos de modernização posam ser concluídos com maior efetividade e fomentar a adoção da ADM.

1.2 Motivação

A primeira motivação deste trabalho é a inexistência de abordagens que dão apoio à criação de motores de transformação que geram modelos PSM a partir de modelos KDM. Essa escassez dificulta a criação de motores desse tipo para dar suporte a projetos de modernização conduzidos com base na filosofia pregada pela ADM/OMG e como consequência dificulta a disseminação dessa filosofia e a adoção na prática.

A segunda motivação é mais técnica e consiste na ausência de motores de transformação do modelo KDM para o modelo Java. Considerando que o KDM é um metamodelo para padronizar a representação de informações no contexto de modernizações, muitas são as situações em que é necessário a transformação de instâncias do KDM em instâncias de algum outro metamodelo. Isso pode ser observado em trabalhos existentes. Por exemplo, transformação do Modelo KDM para o Modelo de Diagramas de Sequência UML (MARTINEZ; PEREIRA; FAVRE, 2014), Modelo KDM para Modelo SBVR (MORATALLA et al., 2012), Modelo KDM para Modelo de Banco de Dados *Database Schema* (PÉREZ-CASTILLO et al., 2009) e Modelo KDM para Modelo de Métricas (CANOVAS; MOLINA, 2010).

Os processos de modernização ADM enfrentam como um dos principais problemas a falta de suportes ferramentais que realizem transformações automáticas entre modelos. Isto é, transformações M2M entre os diferentes níveis de abstração. Em consequência, os projetos de modernização ADM apresentam dificuldade para serem completados e os benefícios fornecidos pelos princípios dirigidos por modelos não podem ser totalmente aproveitados.

1.3 Objetivos

Este trabalho possui os seguintes objetivos:

1. Apresentar uma abordagem iterativa e incremental para a construção de motores de transformação de forma que possa ser adaptada e reusada por outros engenheiros de modernização que tenham interesse em desenvolver transformações de KDM para outros modelos específicos de plataforma (PSM).

2. Disponibilizar uma versão inicial de um motor de transformações de KDM para Modelo Java chamado RUTE-K2J (Rule-based transformarion engine KDM2JAVA) que possa ser estendida e continuada por interessados.

1.4 Contribuições

As principais contribuições que são resultado deste projeto de pesquisa são:

1. Uma abordagem para guiar engenheiros de modernização na construção de motores de transformação a partir do KDM para qualquer PSM, contribuindo com os projetos de modernização para completar o estágio de engenharia avante da ferradura ADM. A abordagem apresentada é dividida em quatro fases e o desenvolvimento do motor de transformação utiliza como principal fonte de informação o mapeamento entre os metamodelos KDM e PSM;

2. Um motor de transformação chamado RUTE-K2J que facilita a transformação exógena do modelo KDM para o Modelo Java e que pode ser utilizado em diversos casos de uso de modernização ADM;

3. O conjunto de regras de transformação ATL do projeto RUTE-K2J disponibilizado para ser continuado e estendido por interessados. O código se encontra armazenado no repositório GitHub e pode ser acessado no seguinte URL:<https://github.com/Advanse-Lab/RUTE-K2J>;

4. As correções feitas nas transformações ATL do *Discover KDM* da ferramenta Modisco (ECLIPSE, 2006b) que foram enviadas e aceitadas pelo projeto Modisco de Eclipse;

5. O conjunto de regras de transformação ATL do projeto *DiscoverKDM-Advanse* que permite gerar um modelo KDM refinado. Embora as correções no *Discover KDM* foram enviadas para o projeto Modisco de Eclipse, ainda serão disponibilizadas em uma próxima versão. O *DiscoverKDM-Advanse* inclui as correções enviadas (e por enviar) dos defeitos encontrados no modelo KDM gerado pela ferramenta Modisco. O código armazenado no repositório GitHub pode ser acessado no seguinte URL: <https://github.com/Advanse-Lab/Discover-Advanse>;

1.5 Cenário de Uso da Ferramenta RUTE-K2J

A Figura 1.1 mostra, além de outros detalhes, um cenário em que a ferramenta RUTE K2J pode ser usada.

Na figura são apresentadas as atividades que pertencem a um processo de Modernização. As primeiras atividades não pertencem ao escopo deste trabalho, mas estão sendo descrita apenas para fornecer uma visão geral do processo todo. Dessa forma, na fase da *Engenharia Reversa*, a primeira atividade que é feita é a geração do modelo KDM a partir do código fonte do sistema legado, utilizando o apoio da ferramenta Modisco. Depois, na fase de *Restruturação*, o modelo KDM é refatorado ou otimizado com o objetivo de melhorar a estrutura interna do código, mas sem alterar o comportamento externo. Para isso, utiliza-se alguma ferramenta de refatoração. A escolha da ferramenta estará sujeita ao critério do engenheiro de software. A saída dessas atividades é o modelo KDM refatorado.

Em seguida, no estágio de *Engenharia Avante*, a atividade em que o modelo Java é gerado, é a atividade que este trabalho de pesquisa fornece suporte. Aqui, o engenheiro de software tem a possibilidade de utilizar o suporte ferramental RUTE-K2J para gerar o Modelo Java a partir do KDM refatorado da atividade anterior.

Por fim, a geração de código fonte que também não pertence ao escopo deste trabalho, mas que combinado com a saída da RUTE-K2J forneceriam um suporte completo para a última parte do estágio de engenharia avante do processo de modernização. Essa última atividade tem como objetivo gerar código fonte Java a partir do Modelo Java. Para isso, considera-se o apoio da ferramenta Acceleo (ECLIPSE, 2006a). O Acceleo considera como núcleo das transformações os *templates*, que são scripts que definem a estrutura do código a ser gerado e que tem como principal vantagem a customização na confecção. Porém existe um plug-in disponível denominado 'Java Generation', que possui templates geradores de código Java já desenvolvidos.

1.6 Organização do Trabalho

Esta dissertação de Mestrado está estruturada da seguinte forma.

No Capítulo 2 apresenta-se a fundamentação teórica para o entendimento do trabalho de pesquisa. No Capítulo 3 são apresentados os trabalhos relacionados, assim como uma discussão das semelhanças e diferenças de cada um deles em relação com a abordagem proposta. No Capítulo 4 é apresentada a Abordagem para a criação de motores de transformação KDM2PSMs, isto é, a descrição em detalhe das fases, atividades e artefatos que compõem a abordagem. No Capítulo 5 é apresenta RUTE-K2J motor de transformação que permite a transformação exógena do modelo Java a partir do modelo KDM. No Capítulo 6 é apresentada a avaliação do motor de transformação RUTE-K2J. Por fim, no Capítulo 7 são apresentadas as conclusões e os trabalhos futuros.

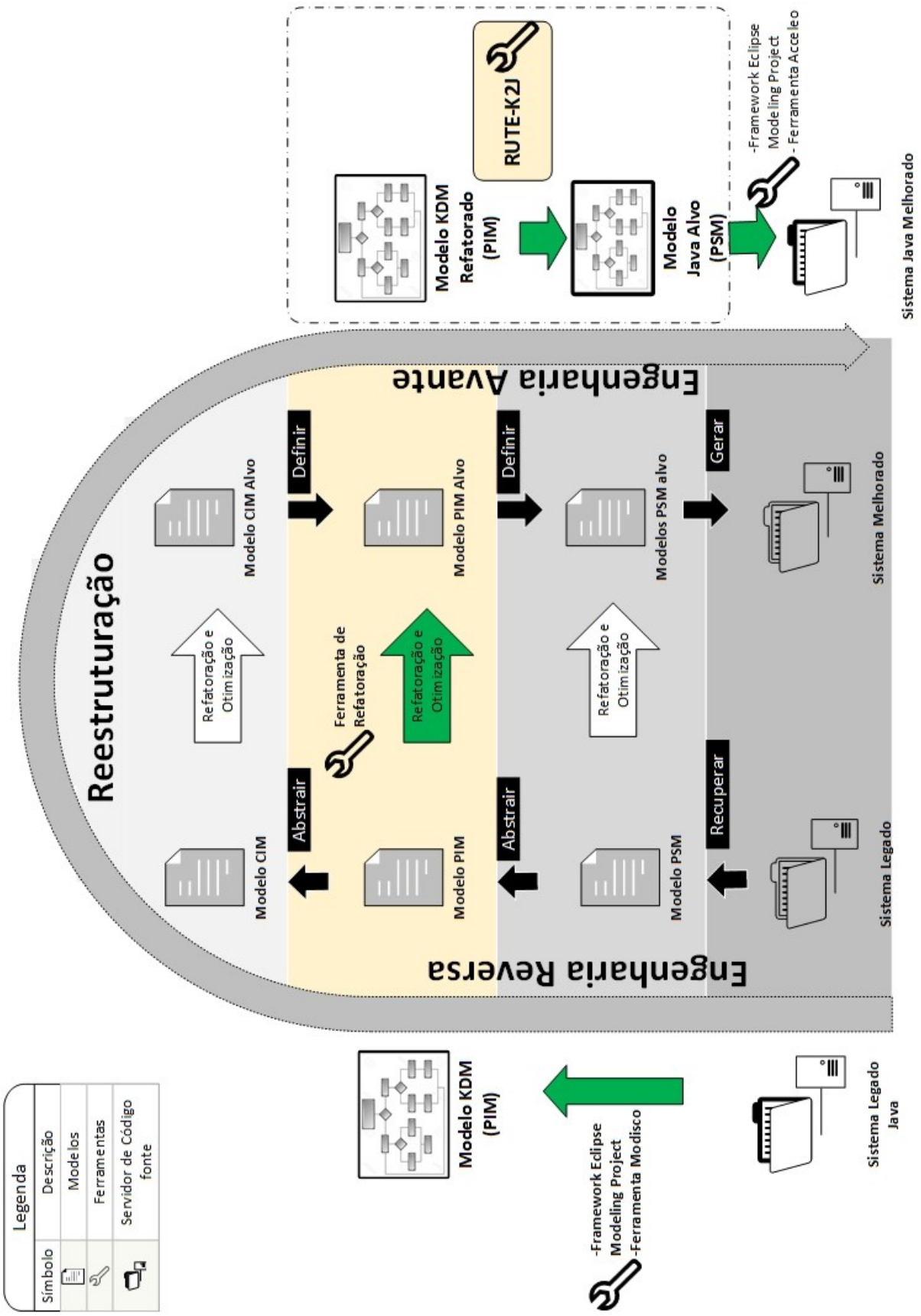

Figura 1.1 – Cenário de Uso

Capítulo 2
FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações Iniciais

O trabalho realizado e apresentado nesta dissertação se desenvolve no contexto de modernização dirigida à arquitetura, especificamente no estágio da engenharia avante. Assim, este capítulo apresenta os conceitos básicos que fazem parte desse contexto.

O capítulo está organizado da seguinte forma: na Seção 2.2 é apresentada a definição, benefícios e padrões da Modernização Dirigida a Arquitetura (ADM). Na Seção 2.3 o Metamodelo KDM e suas caraterísticas são descritas. Na Seção 2.4 são apresentados os tipos de transformações: Endógenas, Exógenas, Horizontais, Verticais, Modelo para Modelo (Model to Model (M2M)) e Modelo para Texto (Model to Text (M2T)). Na Seção 2.5 é apresentada a ferramenta Modisco. Por fim, na Seção 2.6 são apresentadas as considerações finais.

2.2 Modernização Dirigida a Arquitetura - ADM

ADM defende a realização de processos de reengenharia seguindo o padrão MDA e pode ser considerada um mecanismo para a evolução do software. Além disso, permite modernizar e erradicar, ou pelo menos minimizar, os problemas de erosão do software em sistemas legado. Segundo o OMG, a ADM é o processo de compreensão e evolução dos ativos de software existentes, restaurando o valor das aplicações existentes. ADM resolve o problema da reengenharia tradicional, uma vez que realiza processos de reengenharia levando em consideração os princípios orientados por modelo. No entanto, a ADM não substitui a reengenharia, mas a melhora. O padrão proposto pelo OMG define dois princípios: (i) modelar todos os artefatos como modelos em diferentes níveis de abstração; e (ii) estabelecer a transformações entre eles (PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2010).

O modelo de ferradura da reengenharia foi adaptado para a ADM e é conhecido como o Modelo de Modernização em ferradura como mostrado na Figura 2.1. Na ferradura observa-se os três tipos de modelos que são descritos a seguir:

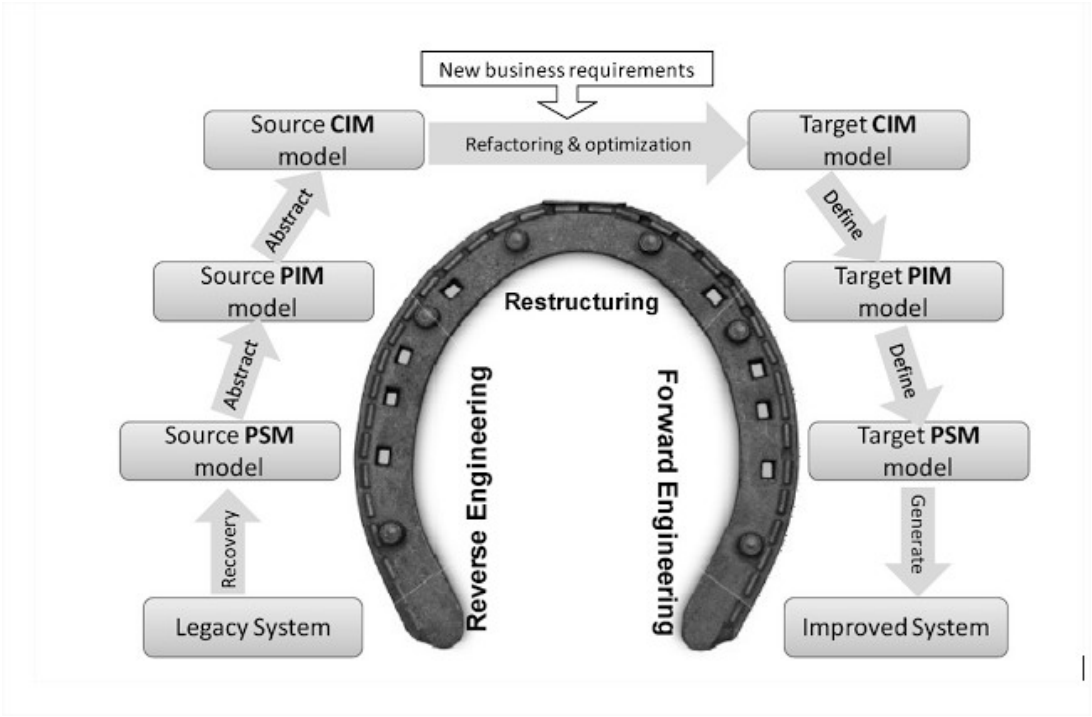

Figura 2.1 – Modelo de modernização em Ferradura (PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2010).

- Modelo Independente de Computação (CIM): que é uma visão do sistema a partir do ponto de vista independente de computação em um alto nível de abstração. Um CIM não mostra detalhes da estrutura do sistema. Os modelos CIM são chamados de modelos de domínio e desempenham o papel de ponte entre os especialistas de domínio e os expertos no design e construção de sistemas.

- Modelo independente de Plataforma (PIM): que é uma visão do sistema do ponto de vista independente da plataforma em um nível de abstração intermediário. Um PIM tem um grau específico de independência tecnológica para ser adequado para uso com uma série de diferentes plataformas de um tipo similar.

- Modelo específico de Plataforma (PSM): que é uma visão de um sistema a partir da plataforma específica em um baixo nível de abstração. Um PSM combina as especificações no PIM com os detalhes que especificam como esse sistema usa um tipo específico de plataforma ou tecnologia.

Além disso, na Figura 2.1, observa-se que o modelo de modernização em ferradura consiste em três etapas: (i) a engenharia reversa identifica o componente do sistema e suas inter-relações e constrói uma ou mais representações abstratas do sistema legado; (ii) a reestruturação/optimização transforma a representação abstrata em outra representação do sistema no mesmo nível de abstração, o que melhora o sistema legado fonte, mas preserva o comportamento externo do sistema; finalmente (iii) a engenharia avançada gera implementações físicas do sistema meta em um baixo nível de abstração.

2.2.1 Vantagens do uso da ADM

Gourshettiwar, Bhandarianish e Shirbhate (2014) descrevem as vantagens que se podem obter usando a ADM, que são: ativar a agilidade do negócio criando agilidade do software; melhorar o retorno do investimento (Return on Investment (ROI)) em softwares existentes, isto é, melhora a produtividade no desenvolvimento de software e reduz o esforço de manutenção e custo; facilitar a adoção de novas tecnologias, práticas e paradigmas. A modernização é feita baseada em experiências anteriores e com as melhores práticas; padronizar a modernização que pode levar a integração e interoperabilidade entre diferentes fornecedores e ferramentas; incentivar a colaboração entre fornecedores complementares; reduzir custos operacionais para os usuários finais; e reduzir tempo, custo e risco das transformações de software.

2.2.2 Padrões ADM

Um dos principais objetivos da ADM é definir um conjunto de metamodelos padronizados para lidar com diferentes desafios que são encontrados na reengenharia de software. Dessa forma, em novembro de 2003, a *Architecture-Driven Modernization Task Force (ADMTF)* criou uma Request for proposal (RFP), que, por sua vez descrevia um conjunto de metamodelos (OMG, 2009). Tais metamodelos são:

- Metamodelo de Descoberta de Conhecimento (KDM). O KDM estabelece um metamodelo inicial que permite que ferramentas de modernização troquem metadados de aplicativos entre aplicativos, linguagens, plataformas e ambientes. Esse metamodelo inicial fornece uma visão abrangente da estrutura e dos dados do aplicativo, mas não representa o software abaixo do nível do procedimento. O KDM estabelece a base para os padrões subsequentes do ADM. O KDM foi adotado em 2006.

- Metamodelo de Árvore sintática abstrata - (Abstract Syntax Tree Metamodel (ASTM)). Este ASTM baseia-se no KDM para representar software abaixo do nível processual. Esse esforço permite que o KDM represente totalmente os aplicativos e facilite a troca de metadados granulares em várias linguagens.

- Metamodelo de Métricas Estruturadas - (Structured Metrics MetaModel (SMM)). O foco é derivar métricas do KDM que podem descrever vários atributos do sistema. Essas métricas transmitem questões técnicas, funcionais e arquitetônicas para os dados e os aspectos processuais dos aplicativos de interesse. Essas métricas suportam o planejamento e a estimativa, a análise de ROI e a capacidade dos analistas de manter a qualidade dos aplicativos e dos dados.

- Reconhecimento do padrão ADM - (ADM Pattern Recognition (ADMPR)). O reconhecimento de Padrões facilita o exame de metadados estruturais com a intenção de derivar padrões e antipadrões sobre os sistemas existentes. Esses padrões e antipadrões podem

ser usados para determinar os requisitos e oportunidades de refatoração e transformação que poderiam ser aplicados a um ou mais sistemas em benefício da empresa.

- Especificação de visualização ADM - (ADM visualization Specification (ADMVS)). A Visualização do ADM se concentra em maneiras de descrever os metadados de aplicativos armazenados no KDM. Isso pode incluir qualquer variedade de visões que possam ser apropriadas ou úteis para planejar e gerenciar iniciativas de modernização. Exemplos incluem o uso de gráficos ou tabelas, resumos de métricas ou modelos de desenvolvimento padronizados.

- Especificação de refatoração ADM - (ADM Refactoring Specification (ADMRS)). A Refatoração do ADM define as maneiras pelas quais o KDM pode ser usado para refatorar aplicativos. Isso inclui estruturação, racionalização, modularização e, de outras formas, aprimoramento de aplicativos existentes sem reprojetar esses sistemas ou, de outra forma, derivar visualizações guiadas por modelos desses sistemas

2.3 Metamodelo de Descoberta de Conhecimento

O Metamodelo de Descoberta de Conhecimento (KDM) fornece um formato de intercâmbio comum que permite a interoperabilidade entre ferramentas de análise e ferramentas de modernização de software, serviços e seus respectivos modelos. Mais especificamente, (KDM) define uma ontologia comum e um formato de intercâmbio que facilita a troca de dados contidos em modelos de ferramentas proprietários que representam o software existente. O metamodelo representa os elementos físicos e lógicos do software, bem como seus relacionamentos em vários níveis de abstração (OMG, 2016).

Ulrich e Newcomb (2010) descrevem que o metamodelo: representa os principais artefatos do software existente como entidades, relacionamentos e atributos; mapeia artefatos externos com os quais o software interage por meio de outros padrões e metamodelos; constitui um núcleo independente da plataforma, mas é extensível para suportar outras linguagens e plataformas; define uma única terminologia unificada para a descoberta de conhecimento dos ativos de software existentes; utiliza um formato de intercâmbio XML Metadata Interchange (XMI), para importar e exportar metadados de ferramentas específicas; facilita o rastreamento de artefatos a partir das estruturas lógicas para os artefatos físicos; e é restrito aos artefatos do ambiente de software existente e representando usando diagramas de classe UML.

O KDM está organizado em 4 camadas e cada camada é organizada em pacotes. Assim, cada pacote define um conjunto de elementos de meta-modelo cuja finalidade é representar uma certa faceta independente do conhecimento relacionado aos sistemas de software existentes. A Figura 2.2 apresenta as camadas, as quais são descritas a seguir:

- Camada de infraestrutura (Infrastructure Layer), que contém os seguintes pacotes: o pacote *Core* define as abstrações básicas do KDM; o pacote *kdm* fornece contexto comparti-

lhado para todos os modelos KDM; o pacote *Source* define elementos de metamodelo que representam o inventário dos artefatos físicos do sistema de software existente e define o mecanismo de rastreabilidade entre os elementos do KDM e sua representação original no *código fonte* do sistema de software existente.

- Camada de Elementos do Programa (Program Elements Layer), que contém os seguintes pacotes: o pacote *Code* define elementos de metamodelo que representam os blocos de construção de software de baixo nível como por exemplo: procedimentos, tipos de dados, classes e variáveis. Dentre os elementos do pacote se encontram: ClassUnit, MethodUnit, StorableUnit, InterfaceUnit, PrimitiveType, entre outros; o pacote *Action* é focado nas descrições de comportamento e as relações de controle e fluxo de dados determinadas por eles.

- Camada de Recursos de Tempo de Execução (Runtime Resource Layer), que contém os seguintes pacotes: o pacote *Plattform* define elementos de metamodelo que representam os recursos de tempo de execução usados pelo sistema de software, bem como os relacionamentos determinados pela plataforma de tempo de execução; o pacote *UI* define os elementos do metamodelo que representam os aspectos da interface do usuário do sistema de software; o pacote *Event* define elementos de metamodelo que representam aspectos orientados a eventos do sistema de software, como eventos, estados, transições de estados, bem como relacionamentos determinados pela semântica baseada em eventos do quadro de tempo de execução; o pacote *Data* define elementos de metamodelo que representam aspectos de dados persistentes do sistema de software.

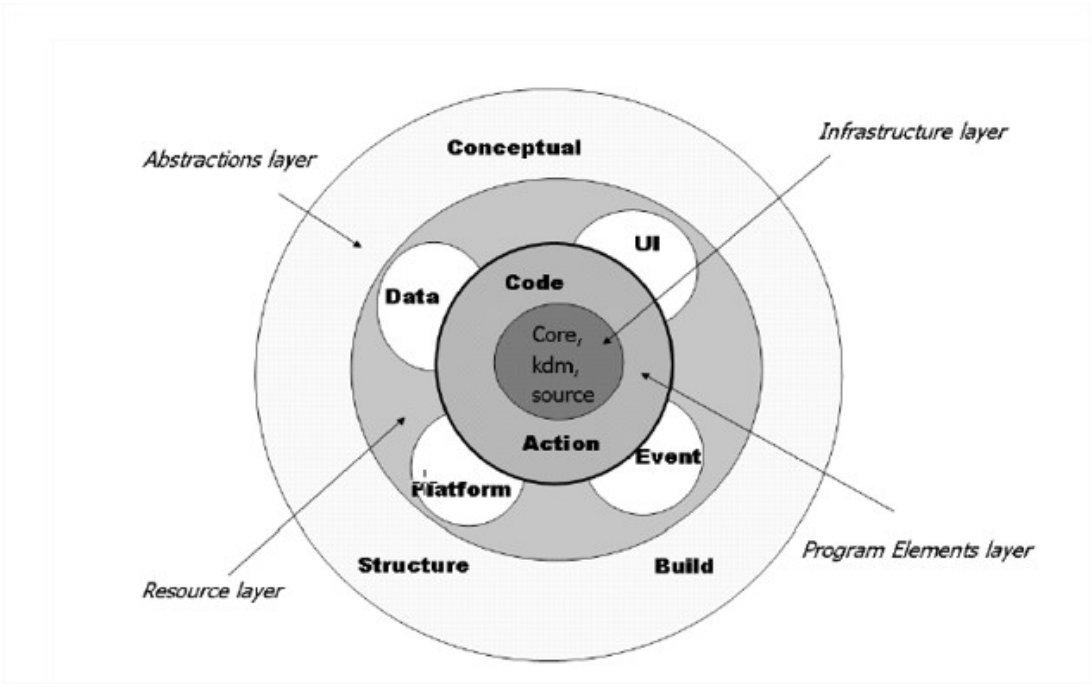

Figura 2.2 – Camadas, pacotes e separação de interesses no KDM (OMG, 2016)

- Camada de abstração (Abstractions Layer), que contém os seguintes pacotes: o pacote *Structure* define elementos de metamodelo que representam componentes arquitetônicos de sistemas de software existentes como subsistemas, camadas, pacotes, entre outros e define a rastreabilidade desses elementos para outros fatos do KDM para o mesmo sistema; o pacote *Conceptual* define elementos de metamodelo que representam os elementos específicos do domínio do sistema de software; o pacote *Build* define elementos de metamodelo que representam os artefatos relacionados ao processo de compilação do sistema de software.

O KDM tem como principal objetivo, fornecer a capacidade de trocar modelos entre ferramentas e, assim, facilitar a cooperação entre fornecedores de ferramentas para integrar vários fatos sobre um aplicativo corporativo complexo, pois a complexidade desses aplicativos envolve múltiplas tecnologias de plataforma e linguagens de programação. Para obter interoperabilidade e integração de informações sobre diferentes facetas de um aplicativo corporativo a partir de várias ferramentas, o metamodelo KDM define vários níveis de conformidade, aumentando assim a probabilidade de que duas ou mais ferramentas compatíveis suportarão os mesmos ou subconjuntos compatíveis de metamodelos.

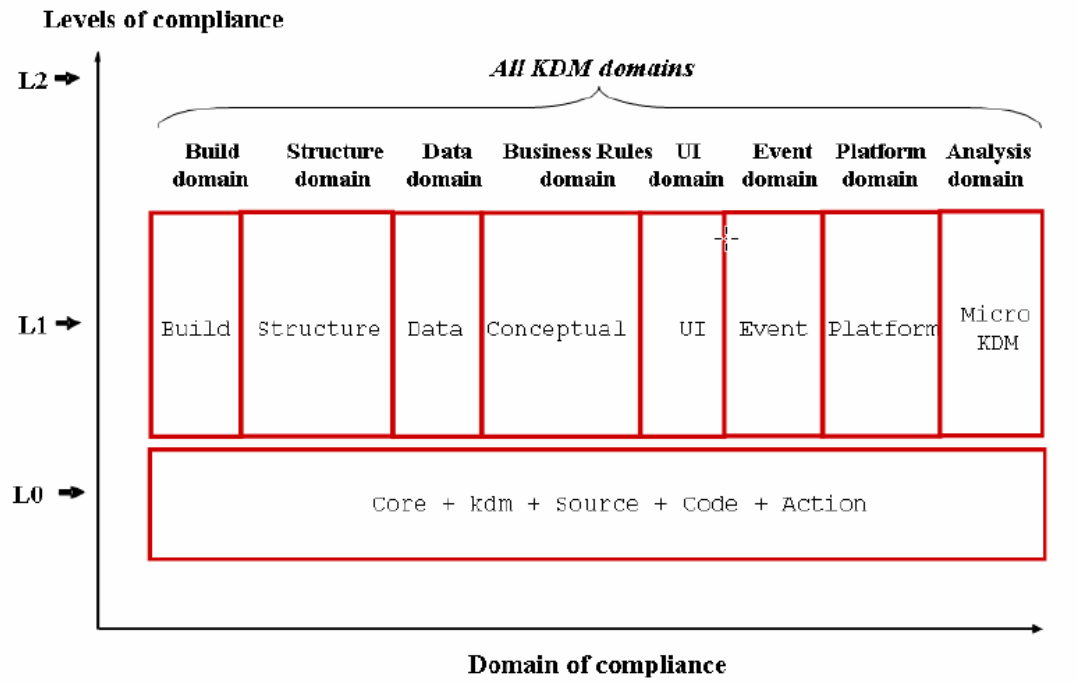

Figura 2.3 – Pacotes e nível de conformidade do metamodelo KDM (OMG, 2016).

O KDM segue o princípio da separação de interesses para permitir a seleção apenas das partes do metamodelo que são de interesse direto para um determinado fornecedor de ferramentas. A separação de interesses no design do KDM é incorporada no conceito de domínios KDM, que define um ponto de vista arquitetônico ISO 42010. A Figura 2.3 apresenta os Domínios

do KDM, os quais sao: Inventory, Code, Build, Structure, Data, Business Rules, UI, Event, Platform, and micro KDM.

Do ponto de vista do usuário, esse particionamento do KDM significa que eles precisam apenas se preocupar com as partes do KDM que consideram necessárias para suas atividades. Se essas necessidades mudarem com o tempo, outros domínios do KDM podem ser adicionados ao repertório do usuário, conforme necessário. Portanto, um usuário do KDM não precisa conhecer o metamodelo completo para usá-lo efetivamente. Além disso, a maioria dos domínios do KDM é particionada em vários incrementos, cada um adicionando mais recursos de conhecimento aos anteriores. Essa decomposição refinada serve para tornar o KDM mais fácil de aprender e usar (OMG, 2016).

2.4 Transformação de Modelos

De acordo com Kleppe, Warmer e Bast (2003), uma transformação é a geração automática de um modelo alvo a partir de um modelo fonte, de acordo com uma definição de transformação. Uma definição de transformação é um conjunto de regras de transformação que juntas descrevem como um modelo na linguagem de origem pode ser transformado em um modelo na linguagem de destino. De forma complementar, uma regra de transformação é definida como uma descrição de como uma ou mais construções na linguagem de origem podem ser transformadas em uma ou mais construções na linguagem de destino.

2.4.1 Transformações endógenas e exógenas

Para transformar modelos, esses modelos precisam ser expressados em alguma linguagem de modelagem. A sintaxe e a semântica da linguagem de modelagem são expressas por um metamodelo. Por exemplo, a sintaxe da *Unified Modeling Language (UML)* é expressada usando diagramas de classes, enquanto a semântica é descrita por uma mistura de regras bem formadas. Com base na linguagem na qual os modelos fonte e alvo de uma transformação são expressados, pode-se fazer uma distinção entre transformações endógenas e exógenas. As transformações endógenas são transformações entre modelos expressados na mesma linguagem enquanto as transformações exógenas são transformações entre modelos expressos em diferentes linguagens (MENS; GORP, 2006).

Visser (2001) propõe a mesma distinção, utilizando o termo *reformulação* para uma transformação endógena, enquanto o termo *tradução* é usado para uma transformação exógena. Esses cenários principais podem ser refinados em vários subcenarios típicos com base no efeito no nível de abstração do programa e até que ponto preservam a semântica dele. Na Figura 2.4 apresenta-se a taxonomia, que é descrita a seguir.

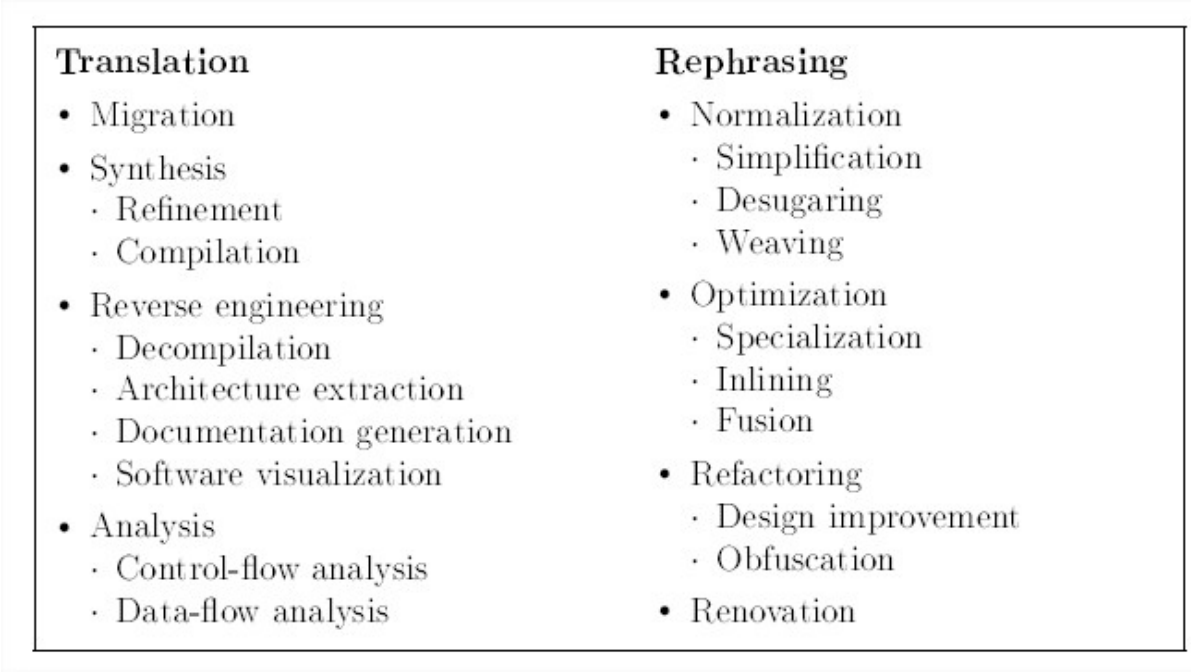

Figura 2.4 – Taxonomia de transformação de programas (VISSER, 2001)

2.4.1.1 Tradução

Em um cenário de tradução, um programa em uma linguagem de origem é transformado para um programa em uma linguagem de destino diferente. Os cenários de tradução podem ser distinguidos pelo seu efeito no nível de abstração de um programa. Embora as traduções tenham como objetivo preservar a semântica extensional de um programa, geralmente não é possível reter toda a informação através de uma tradução. Os cenários de tradução podem ser divididos em síntese, migração, engenharia reversa e análise (VISSER, 2001). Na Figura 2.5 é apresentada os tipos de cenários de Tradução.

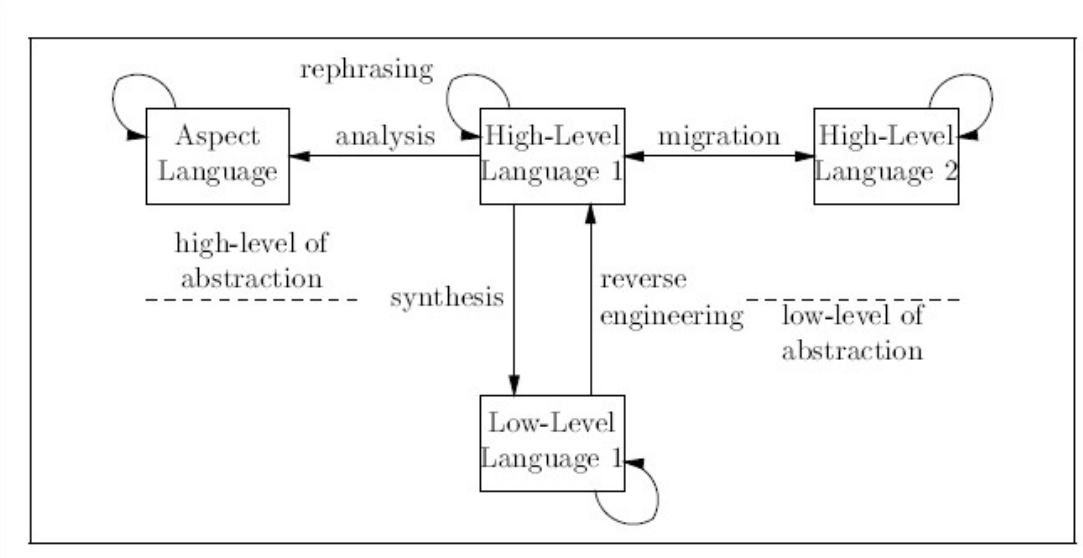

Figura 2.5 – Relação entre os tipos de transformação de programas (VISSER, 2001)

A *síntese* de um programa é um tipo de transformação que reduze o nível de abstração de um programa. Durante o processo de síntese, as informações de projeto são negociadas para aumentar a eficiência. O refinamento permite que uma implementação seja derivada de uma especificação de alto nível, de modo que a implementação satisfaça a especificação. De igual forma, a compilação é um tipo de síntese em que um programa em uma linguagem de alto nível é transformado em código máquina.

Na *migração*, um programa é transformado em outra linguagem no mesmo nível de abstração. Isto pode ser uma tradução entre dialetos, por exemplo transformar um programa *Fortran77* em um programa *Fortran90* equivalente; ou uma tradução de uma linguagem para outra, por exemplo transferir um programa *Pascal* para *C*.

A *engenharia reversa* tem como objetivo elevar o nível de abstração, isto é, extrair a partir de um programa de baixo nível, um programa ou especificação de alto nível ou pelo menos alguns aspectos. Exemplos de engenharia reversa são: a descompilação, em que um programa objeto é traduzido em um programa de alto nível; extração de arquitetura; geração de documentação; e visualização de software, em que algum aspecto de um programa é representado de forma abstrata.

A *análise* reduz um programa a um aspecto como fluxo de controle ou de dados, pode ser considerada uma transformação para uma sublinguagem.

2.4.1.2 Reformulação

A reformulação transforma um programa em um programa diferente na mesma linguagem, isto é, a linguagem de origem e de destino são as mesmas. Em geral, as reformulações tentam dizer a mesma coisa com diferentes palavras visando melhorar alguns aspectos do programa. Na Figura 2.4 observa-se os principais subcenários da reformulação como: normalização, otimização, refatoração e renovação (VISSER, 2001).

A normalização reduz um programa para outro programa em uma sublinguagem com o objetivo de diminuir sua complexidade sintática. *'Desugaring'* é um tipo de normalização em que alguns dos construtores (sintaxes açúcar) de uma linguagem são eliminadas traduzindo-os em construtores mais fundamentais. A simplificação é um tipo mais geral de normalização em que um programa é reduzido a uma forma normal (padrão), sem necessariamente remover construtores simplificados. Por exemplo, considere a transformação em forma canônica de representações intermediárias e simplificação algébricas de expressões.

Otimização é uma transformação que melhora o tempo de execução e/ou desempenho de espaço de um programa. Os exemplos de otimizações são fusão, propagação constante, eliminação de subexpressão comum e eliminação de código morto.

Refatoração é uma transformação que melhora o *design* de um programa, reestruturando-o de tal forma que se torna mais fácil de entender, preservando sua funcionalidade. Ofuscação é uma transformação que torna um programa mais difícil de entender, renomeando variáveis,

inserindo código morto, etc. Ofuscação é feito para ocultar as regras de negócios incorporadas no software, tornando mais difícil a engenharia reversa do programa.

Na Renovação de software, o comportamento extensional de um programa é alterado para reparar um erro ou atualizá-lo em relação a requisitos alterados. Exemplos são a reparação de um bug Y2K (o problema do ano 2000), ou converter um programa para lidar com o Euro.

2.4.2 Transformações horizontais e verticais

Uma transformação horizontal é uma transformação onde os modelos fonte e alvo residem no mesmo nível de abstração. Exemplos típicos são refatoração (uma transformação endógena) e migração (uma transformação exógena). Por outro lado, uma transformação vertical é uma transformação onde os modelos fonte e alvo residem em diferentes níveis de abstração. Um exemplo típico é refinamento, onde uma especificação é gradualmente refinada em uma implementação completa, por meio de sucessivas etapas de refinamento que adicionam mais detalhes concretos (MENS; GORP, 2006). Na Tabela 2.1 é apresentada as dimensões horizontal e vertical, assim como endógena e exógena.

Tabela 2.1 – Dimensões horizontais versus vertical e endógena versus exógena

Tipo	Horizontal	Vertical
Endógena	Refatoração	Refinamento formal
Exógena	Migração de linguagem	Geração de código

2.4.3 Transformações Modelo para Modelo (M2M)

Nas transformações de Modelo para Modelo (M2M) as ferramentas convertem uma ou mais fontes em um ou mais alvo (s). As linguagens de transformação fornecem um conjunto de construções ou mecanismos para aplicar transformações. Os diferentes tipos de abordagens são: relacional, imperativo, baseado em grafos, outras abordagens e híbridos (KAHANI; CORDY, 2015).

Abordagens Relacionais concentram-se no que deve ser transformado sem especificar uma sequência explícita na ordem de execução. Essas abordagens são baseadas na definição de relações entre os elementos nos modelos fonte e alvo e são definidas em relações matemáticas que podem ser especificadas por predicados e restrições.

Abordagens relacionais incluem programação funcional e programação lógica. Uma linguagem lógica tem muitos recursos, como mecanismos de busca, propagação de restrições e *backtracking* que a torna apropriado. Em linguagens funcionais, uma função pode transformar a entrada(s) na saída(s) e tem a vantagem de que o desenvolvedor não precisa lidar com a tarefa não trivial de escrever código para atravessar o modelo. Ferramentas como *UML-RSDS*, *Tefkat*, *JTL*, *PTL*, *ModTransf*, *PETE* e *TXL* são exemplos de abordagens relacionais.

Abordagens imperativas concentram-se em como e quando a transformação deve ser executada, sem levar em conta as relações que devem existir entre os elementos fonte e alvo. A linguagem especifica uma transformação como uma sequência de ações / regras. As linguagens imperativas são semelhantes às linguagens de programação clássicas, por isso são fáceis de trabalhar para os desenvolvedores.

Query/View/Transformation Operational (QVTo) é um exemplo de linguagem imperativa, que é comparável com as linguagens processuais convencionais como *C*. As transformações *QVTo* são definidas usando mapeamentos. Os mapeamento podem transformar um ou mais elementos de um modelo de origem para o(s) elemento(s) alvo correspondente(s). Além disso, os mapeamentos podem conter cláusulas *'when'* e *'where'*. Exemplos de ferramentas imperativas são ModelAnt, Xtend, Kermeta2, Modelio, Umple, MDWorkbench, Melange, Enterprise Architect (EA), WebRatio, Mitra2, JQVT, Merlin e MOFScript.

Abordagens baseadas em Grafos são baseadas em gramáticas de grafos algébricos e representam os modelos de origem e de destino usando variações de grafos tipados, atribuídos e rotulados. A transformação de grafos consiste em um conjunto de regras de reescrita (também chamadas de regras de transformação de grafos ou regras de produção) e um grafo *host* ao qual as regras são aplicadas para criar um novo grafo. Cada regra compreende um grafo do lado esquerdo (Left Hand Side (LHS)) e um grafo do lado direito (Right Hand Side (RHS)). A execução de uma regra em um grafo host envolve todos os elementos que apenas estão no LHS, todos os elementos que estão em RHS, mas não aparecendo em LHS são adicionados e todos os elementos correspondentes que existem em ambos lados permanecem inalterados.

As transformações de grafos possuem sólidas bases teóricas que permitem ser usadas na verificação formal das transformações. O principal inconveniente da notação de grafos é a complexidade e a verbosidade de representar as regras de transformação. Exemplos de ferramentas nesta categoria são: AToMPM, GROOVE, UMLX, AToM3, AGG, BOTL, GRoundTram, GReAT, MOMoT, PROGRES e MoTMoT.

Outra abordagem é de transformações implementadas usando a Linguagem de Estilo (Xtensible Style-sheet Language Transformation (XSLT)), que é uma linguagem padrão para transformar XML e que pode ser aplicado para implementar transformações de modelos. XSLT é uma plataforma independente que usa o conceito de padrões e opera sobre a representação textual de modelos. Essa abordagem percorre a estrutura de árvore XML para localizar os nós da árvore que correspondem ao seu padrão. No caso da correspondência de padrões, o XSLT aplica uma regra de transformação específica. No entanto, tem limitações de escalabilidade, as transformações são complexas e detalhadas. Assim, a manutenção de transformações de modelo implementada em XSLT é difícil. Além disso, o XSLT só pode suportar estruturas de árvore e não grafos arbitrariamente.

Abordagens híbridas cada técnica tem suas próprias forças e fraquezas. Em abordagens imperativas, um programador tem um alto nível de controle sobre a execução da transformação que resulta em uma implementação eficiente de transformações, especialmente para as complexas.

No entanto, o controle explícito pode levar a escrever mais código que torna essa abordagem mais difícil de ler e entender. Enquanto, as linguagens relacionais definem transformações em um nível mais alto de abstração e escondem os detalhes relacionados ao processo de transformação, tornando a tarefa do desenvolvimento da transformação mais fácil, concisa e mais curta. As linguagens de grafos têm problemas de escalabilidade para lidar com modelos grandes. Desta forma, abordagens híbridas, que combinam as forças de mais de uma abordagem de transformação de modelo, podem ser usadas para especificar transformações. VIATRA, Eclética, Epsilon, AGE, Atlas Transformation Language (ATL), Rational e Blu Age são exemplos de ferramentas de transformação híbridas.

2.4.4 Transformações Modelo para Texto (M2T)

Ferramentas de transformação de Modelo para texto (M2T) transformam um ou vários modelos em um fluxo de caracteres em termos de código fonte (por exemplo, C ++, Java) ou outros formulários textuais como arquivos de configuração. Os diferentes tipos de abordagens de transformação são baseados em visitantes, baseado em template e híbrido (KAHANI; CORDY, 2015), (CZARNECKI; HELSEN, 2006).

Abordagens baseadas em visitantes atravessam uma representação interna baseada em árvore de um modelo para gerar código para cada elemento do modelo. O código gerado é escrito em um fluxo de texto. A ordem dos modelos a serem percorridos e o código a gerar são definidos por regras. No entanto, o desenvolvedor tem que desenvolver algumas partes da transformação para enviar o texto para a saída. Exemplos de ferramentas nesta categoria são Kermeta2, Melange, JAMDA, ATOM3 e ATOMPM.

Nas *Abordagens Baseadas em Template*, um *template* define a parte estática da estrutura destino, compartilhada por todos os artefatos e as variáveis, que podem ser substituído por valores dos elementos do modelo de origem. Existe uma metaprogramação para a parte dinâmica, que fornece acessibilidade às informações armazenadas nos modelos.

A similaridade da estrutura do *template* com o código gerado, torna essa abordagem mais precisa e mais fácil de compreender. Além disso, a reutilização dos *templates* torna o processo de desenvolvimento mais simples. Exemplos de ferramentas baseadas em templates são ModelAnt, ModTransf, Umple, Acceleo, MagicDraw, AGE, eMoflon, Henshin, MDWorkbench, AndroMDA, Fujaba, WebRatio, Enterprise Architect (EA), UMT, Merlin, MOFScriptt, Rational, Xpand, VIATRA e Epsilon.

Abordagens híbridas, embora a abordagem baseada em visitantes pareça ser mais fácil, não é adequada quando a maior parte da geração de código consiste em texto estático. Portanto, as linguagens baseadas em templates podem ser combinadas com o padrão visitante para projetar e implementar ferramentas M2T. Actifsource, MetaEdit +, Blu Age, VMTS, Xtend, GrGen.NET e Modelio são ferramentas baseadas em abordagens híbridas.

2.5 Ferramenta Modisco

MoDisco é um projeto de código aberto oficialmente parte da Eclipse Foundation (ECLIPSE, 2006b), que promove técnicas de Engenharia Reversa dirigida por Modelos (Model Driven Reverse Engineering (MDRE)) e seu desenvolvimento dentro da comunidade Eclipse. Atualmente é reconhecido oficialmente e citado pelo OMG como fornecedor de implementações e ferramentas de referência de vários dos padrões da indústria da ADMTF, como o modelo KDM, modelo SMM e modelo ASTM (BRUNELIERE et al., 2014).

MoDisco tem como objetivos oferecer um framework MDRE genérico e extensível de código aberto, assim como fornecer recursos necessários para criar modelos e permitir seu tratamento, análise e computação. Na Figura 2.6, observa-se que Modisco considera como entradas todos os tipos de artefatos legados, por exemplo: código fonte, bancos de dados, arquivos de configuração e documentação. Como saídas, o framework visa a produção de diferentes tipos de artefatos, dependendo do(s) objetivo(s) de engenharia reversa selecionado, por exemplo: código fonte, dados, métricas e documentação.

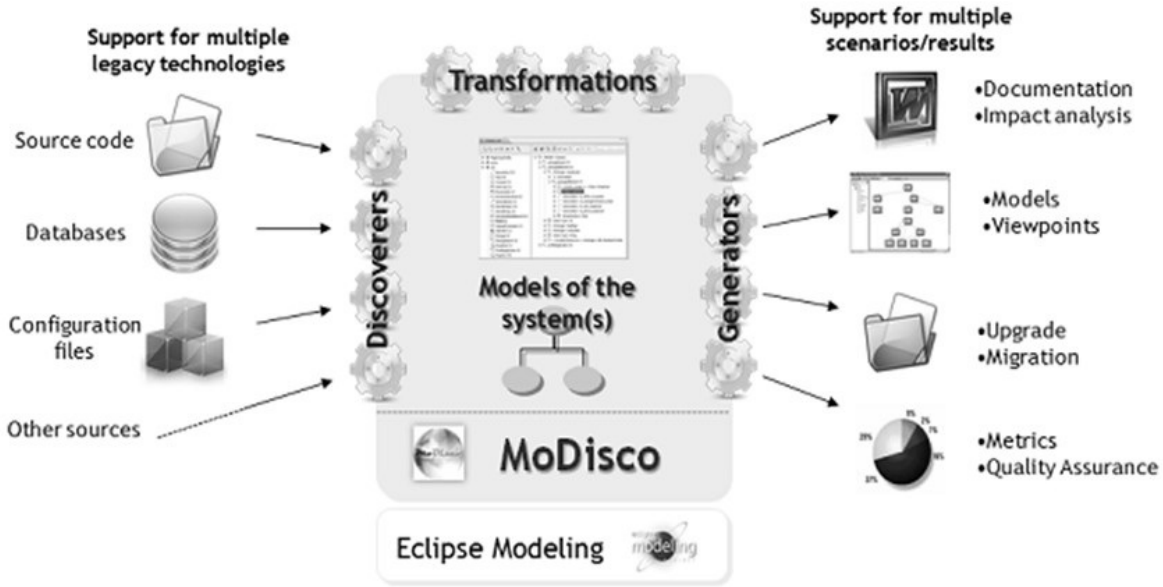

Figura 2.6 – Ferramenta Modisco (BRUNELIERE et al., 2010)

Modisco foi inicialmente criado como um framework de pesquisa experimental pela Equipe AtlanMod (EMN & INRIA), evoluindo para uma solução industrializada graças à colaboração com a empresa MIA-Software. Esse trabalho em conjunto teve como resultado um kit de ferramentas eficiente e utilizável para a descoberta, consulta e manipulação de modelos de software da engenharia reversa.

O framework é equipado com componentes essenciais, genéricos e personalizáveis, como um navegador de modelos, um modelo de extensão e mecanismos de personalização, um gerenciador de consultas de modelo, um gerenciador de descoberta e fluxo de trabalho e algumas

facilidades de visualização de métricas. Também fornece e usa implementações concretas de dois metamodelos padrão do OMG, o Metamodelo de descoberta de conhecimento (KDM) e Metamodelo de métrica de software (SMM), que são metamodelos genéricos (independentes da tecnologia). Além disso, oferece suporte tecnológico específico para engenharia reversa Java (incluindo um metamodelo Java completo, descobridor e transformação correspondentes ao KDM) e engenharia reversa XML (para alguns arquivos de configuração da estrutura JEE, como Struts ou Hibernate) (BRUNELIERE et al., 2010).

2.5.1 Benefícios do MoDisco

Bruneliere et al. (2014) descrevem os principais benefícios da utilizacao do framework Modisco.

Em primeiro lugar, MoDisco permite cobrir diferentes níveis de abstração e satisfazer vários graus de detalhe, dependendo das necessidades do cenário de engenharia reversa. Todas as informações necessárias podem ser representadas como modelos para que não exista perda de informações durante os processos do MDRE (exceto para aqueles detalhes que o usuário explicitamente deseja deixar de fora como parte de um processo de transformação dos modelos descobertos iniciais).

Em segundo lugar, o uso de técnicas MDE permite a decomposição e automação dos processos de engenharia reversa. Eles podem ser divididos em etapas menores com foco em tarefas específicas e ser amplamente automatizados graças ao encadeamento de operações MDE (especialmente transformações de modelo). Todos os artefatos de modelagem envolvidos (modelos, metamodelos, transformações, etc.) podem ser reutilizados de forma homogênea, modificados por motivos de manutenção e evolução ou estendidos para outros fins. Além disso, novas transformações podem ser desenvolvidas e acrescentadas adicionando mais recursos sem alterar os recursos já implementados.

Finalmente, o tratamento da enorme quantidade de dados pode ser simplificado porque os modelos dos sistemas são os elementos realmente processados (graças às técnicas de modelagem disponíveis) e não diretamente os próprios sistemas (que não são modificados durante o processo). O desempenho observado nos principais componentes do MoDisco é aceitável para uso industrial em vários cenários concretos.

2.6 Considerações Finais

Este capítulo proporciona os principais conceitos para o bom entendimento deste trabalho de pesquisa. Dentre os temas apresentados, foram abordados conceitos sobre ADM, vantagens, modelos padrões e o processo de modernização proposto pela ADM/OMG.

Além disso, foi descrito o metamodelo KDM que representa os principais artefatos do software existentes como entidades, relacionamentos e atributos. KDM é o metamodelo utilizado

na abordagem proposta neste trabalho de mestrado.

Também, apresentou-se os tipos de transformações: endógenas, exógenas, horizontais, verticais, Modelo para Modelo (M2M) e Modelo para Texto (M2T), com o objetivo de facilitar a compreensão das transformações realizados no decorrer da abordagem.

Finalmente, foi apresentada a ferramenta Modisco que fornece os recursos necessários para criar modelos a partir de um sistema legado, permitindo sua análise e computação. Essa ferramenta foi utilizada no contexto deste trabalho para gerar os modelos KDM e Java a partir de código fonte.

Capítulo 3

TRABALHOS RELACIONADOS

3.1 Considerações Iniciais

Neste capítulo são apresentados alguns trabalhos relacionados com o trabalho de pesquisa. Muitos dos trabalhos apresentados não fornecem detalhes acerca da etapa de engenharia avante ou mesmo das técnicas de transformação empregadas. Entretanto, é interessante notar que todos eles possuem máquinas de transformação de KDM para algum modelo, seja ele proprietário ou não. Também é importante ressaltar que esses autores também precisaram, de uma forma ou de outra, criar tabelas de mapeamento entre o KDM e os outros modelos, bem como escrever as transformações.

3.2 Detalhamento dos Trabalhos

Moratalla et al. (2012) propõem uma abordagem denominado GAFEMO (Abordagem de Análise de Lacunas que Integram os Princípios da Modernização Orientada a Arquitetura), que apoia o processo de modernização de sistemas legados na adoção dos princípios orientados a serviços por meio de técnicas de análise de lacunas.

O processo consiste em 8 passos, como mostrado na Figura 3.1. Os três primeiros passos referem-se ao estágio de engenharia reversa com o objetivo de criar um modelo KDM completo. Para isto, o modelo KDM é gerado com suporte do *Discover KDM* da ferramenta Modisco e refinado para completar informação faltante, detectada pelos autores, por meio da execução de regras de transformações adicionais.

No quarto passo é realizada a transformação do modelo KDM para o modelo SBVR com suporte de um motor de transformação, que é constituído por um conjunto de regras de transformação desenvolvidas pelos autores. O metamodelo SBVR é um metamodelo promovido pelo OMG que define o vocabulário e as regras para documentar a semântica de vocabulários e regras de negócios facilitando a troca de informação entre organizações e ferramentas de software.

Nos passos número cinco e seis é gerado um novo modelo SBVR chamado de alvo, obtido a partir do modelo SBVR gerado, dos objetivos de negocio estabelecidos e por meio de uma serie

de transformações manuais. Por fim, nos passos sete e oito, no estágio de engenharia avante, o modelo KDM e ASTM são gerados.

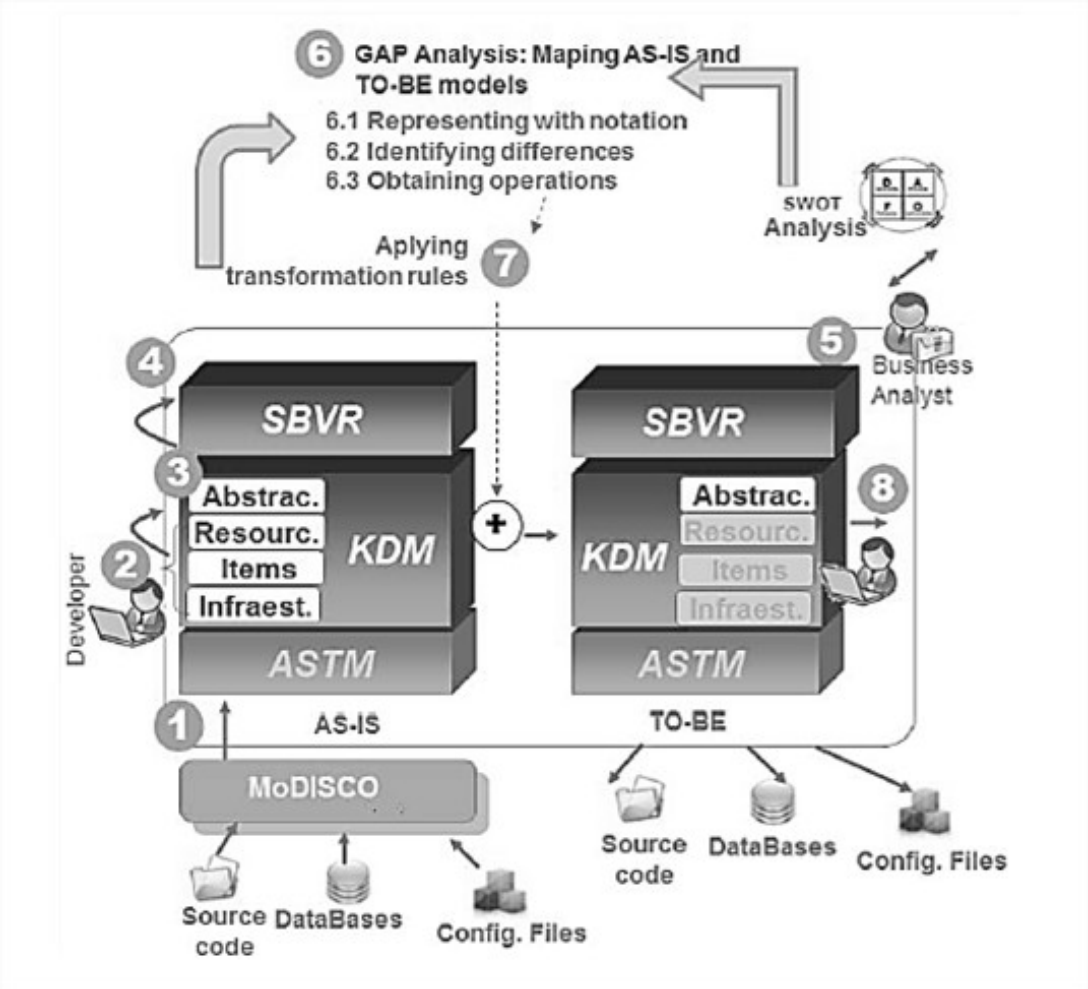

Figura 3.1 – Passos da modernização do framework GAFEMO (MORATALLA et al., 2012)

Na abordagem, a ferradura de modernização não é completada sequencialmente. O modelo KDM (fase de engenharia avante) é gerado por meio de uma transformação horizontal e endógena a partir do modelo KDM completo (fase de engenharia reversa); ao invés de ser uma transformação que segue a ordem da ferradura, isto é, do modelo SBVR alvo para o modelo KDM. Além disso, os autores não oferecem detalhes das transformações do modelo KDM para o modelo SBVR, na fase de engenharia reversa, nem do modelo KDM para o modelo AST, fase de engenharia avante.

Um outro trabalho relacionado a este trabalho é o de Pérez-Castillo et al. (2013) que propõe AndrIU, uma ferramenta que transforma as interfaces de usuário de aplicações *desktop* em interfaces de usuário de aplicativos móveis. AndrIU está baseada na análise estática do código fonte facilitando a migração de sistemas tradicionais para aplicações Android. A Figura 3.2 apresenta as 3 etapas do processo de migração.

Na primeira etapa, o modelo Abstract Syntax Tree (AST), instância do metamodelo Java Swing, é gerado a partir do código fonte. Na segunda etapa é gerado o modelo KDM a partir do

modelo AST por meio da aplicação de regras de transformações QVT. Por fim, na terceira etapa, os autores propõem transformações entre o modelos KDM e modelo *Android* baseadas em um mapeamento entre os metamodelos e desenvolvidas com a ferramenta Medini QVT. Os autores não fornecem maiores detalhes do mapeamento feito entre os elementos dos metamodelos, nem acesso às regras de transformação para análise e aprimoramento.

É interessante notar, que os autores desenvolveram um suporte ferramental que implementa as três etapas descritas e que lida com os metamodelos Java Swing e Android utilizados na abordagem. Essa ferramenta possui internamente três motores de transformação que permitem gerar os modelos intermediários descritos, possibilitando a migração da interface Java Swing.

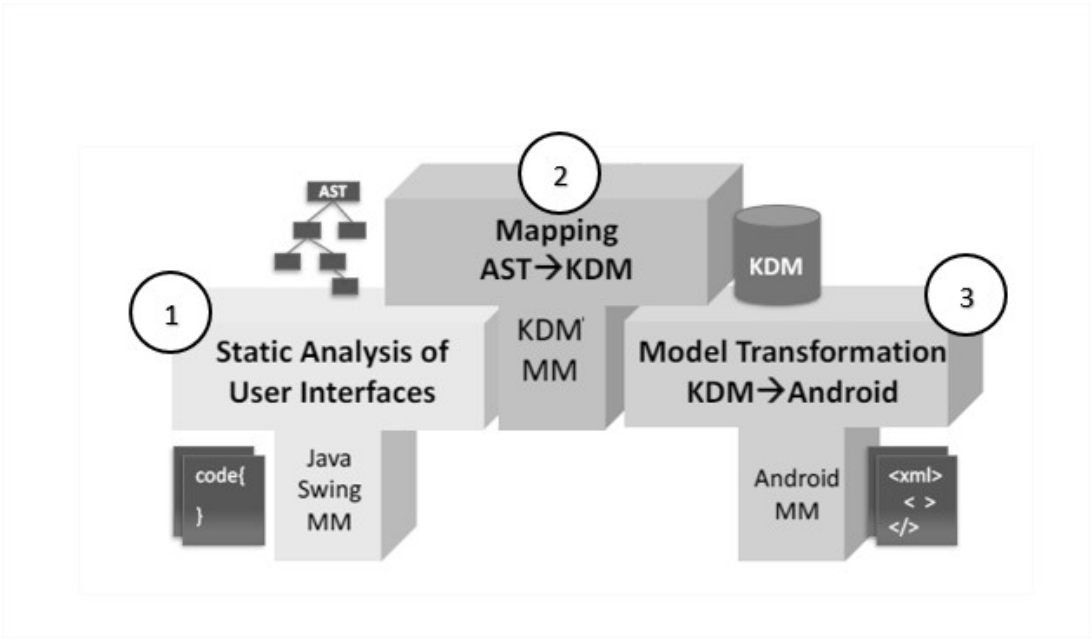

Figura 3.2 – Processo de migração da interface de usuário adaptado de Pérez-Castillo et al. (2013)

Outro trabalho é o de Rodríguez-Echeverría et al. (2011), que propõem a definição de um framework para a modernização de aplicações web (Web Application - WA) em aplicações Clientes que seguem os princípios de Aplicações de Internet enriquecidas (Rich Internet Applications (RIA)). O processo de modernização consiste em 4 fases principais como é ilustrada na Figura 3.3.

As duas primeiras fases estão focadas na obtenção do modelo KDM. Para isto, primeiro, é obtido o modelo ASTM com suporte da ferramenta Modisco. Seguidamente, é gerado o modelo KDM a partir do modelo ASTM por meio de transformações baseadas na análise estática e dinâmica do código fonte e Logs do sistema.

Na terceira fase, chamada de *Projetar Plataforma*, é feita a transformação do modelo KDM para o modelo RIA-extended MDWE, os autores não fornecem maiores detalhes da transformação realizada. O metamodelo RIA-extended MDWE é uma extensão do metamodelo MDWE e enriquecido com as características da tecnologia RIA, recopiladas pelos autores a partir de

outras abordagens que incluem essa tecnologia. Por fim, a última fase consiste na geração do código fonte com suporte de ferramentas M2T, como JET e Xpand.

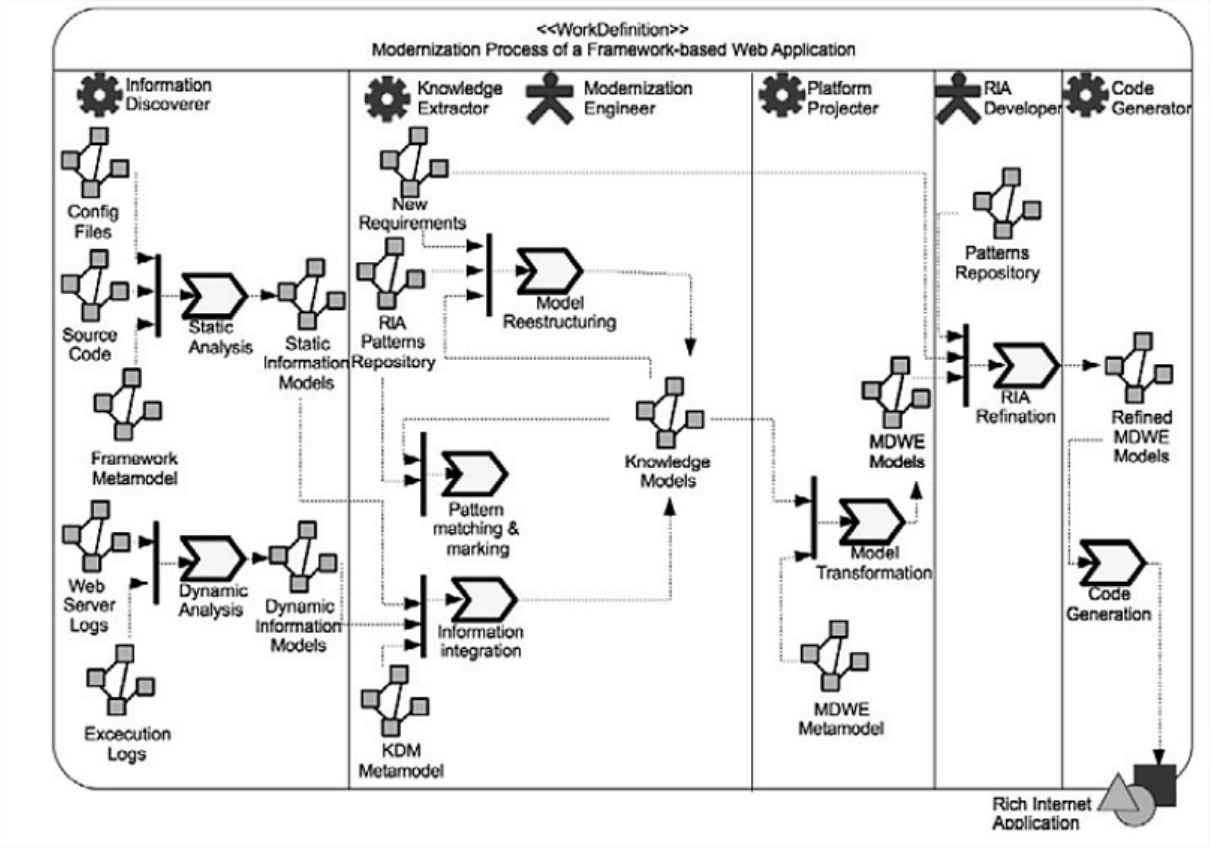

Figura 3.3 – Visão geral do processo de modernização (RODRÍGUEZ-ECHEVERRÍA et al., 2011)

O interessante na abordagem é que os modelos intermediários construídos ao longo da abordagem estão baseados em informações estáticas, dinâmicas, definição de novos requisitos e padrões RIA estabelecidos, embora não seja descrita como foi realizada essa integração. Além disso, o motor de transformação, as regras de transformação e o mapeamento entre os elementos dos metamodelos KDM refinado para o modelo RWA-extended MDWE não são descritos.

Um outro trabalho relacionado a este trabalho é o de Martinez, Pereira e Favre (2014), que apresentam um framework de engenharia reversa dirigida pela arquitetura (ADRE) para recuperar diagramas de sequência UML a partir do código orientado a objetos. O processo de engenharia reversa consiste em duas etapas principais descritas abaixo e mostradas na Figura 3.4.

A primeira fase denominada de descoberta do modelo, permite a transformação do código fonte em modelos AST utilizando o descobridor (Discoverer) javaAST do Modisco. Na segunda fase denominada de entendimento do Modelo, os autores propõem duas transformações M2M, com o objetivo de gerar o modelo KDM e o modelo de diagrama de sequência UML.

A primeira transformação consiste na geração do modelo KDM a partir do modelo AST. Para isso, utiliza-se o descobridor do KDM fornecido pela ferramenta Modisco. Entretanto, como o descobridor KDM do Modisco possui algumas limitações em termos de informações

recuperadas, os autores fizeram algumas evoluções nesse Descobridor para que ele pudesse recuperar todas as informações necessárias. A segunda transformação consiste na geração do diagrama sequência a partir do modelo KDM. Para isto, é realizado um mapeamento entre os elementos dos metamodelos KDM - UML. Depois, são desenvolvidas regras de transformação baseadas nesse mapeamento. Por fim, as regras de transformação desenvolvidas são aplicadas no modelo KDM gerando como resultado o modelo de sequência UML.

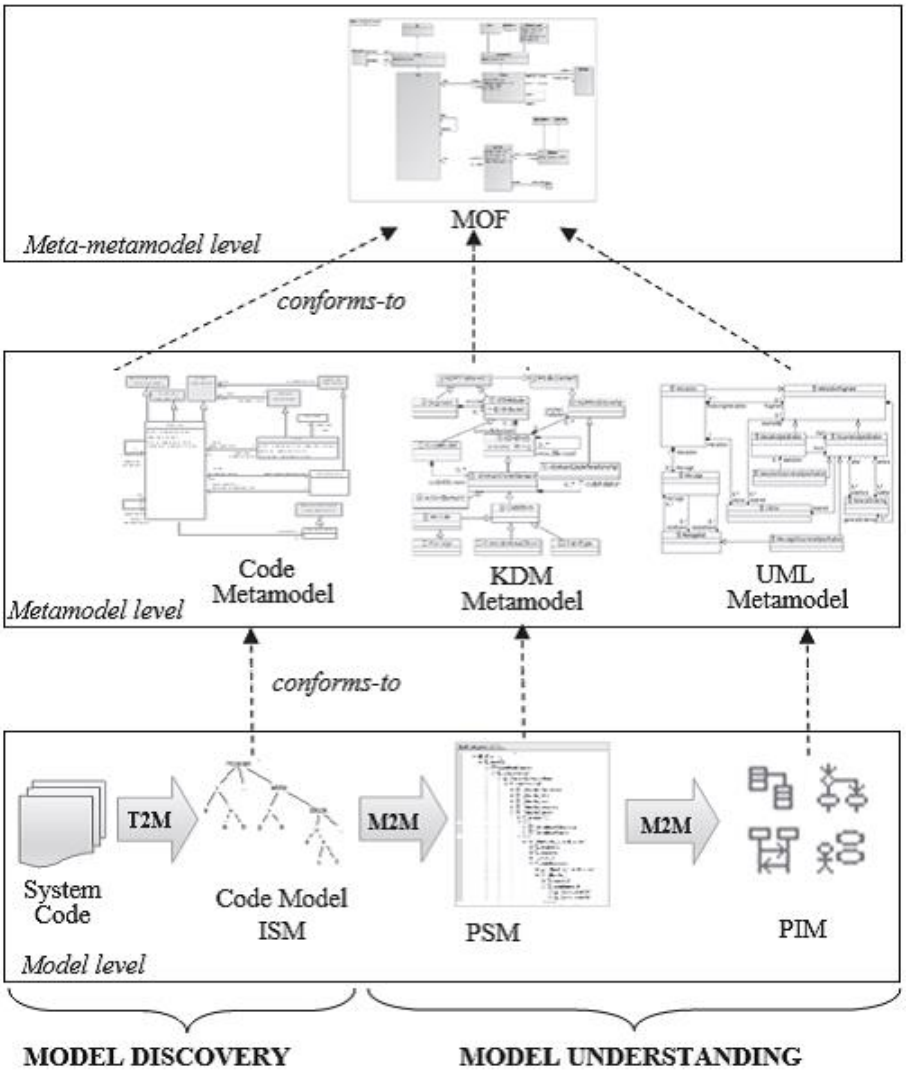

Figura 3.4 – ADRE Framework (MARTINEZ; PEREIRA; FAVRE, 2014)

O trabalho relacionado apresentado compreende o estágio de engenharia reversa. Nele as limitações da ferramenta Modisco são detectadas e minimizadas com o desenvolvimento de regras de transformação adicionais. Além disso, as regras de transformação principais que conformam o motor de transformação KDM para UML são descritas em linguagem natural e parte do código ATL desenvolvido é apresentado fornecendo uma visão das metaclasses KDM utilizadas.

Um outro trabalho relacionado a este trabalho é o de Pérez-Castillo et al. (2009), que propõem uma técnica de engenharia reversa denominada de *Contextualização de Dados* e uma

ferramenta que dá suporte a sua aplicação. Essa técnica recupera fragmentos de código fonte legado e os trechos de sentenças de banco de dados embutidas neles permitindo seu gerenciamento no processo de reengenharia. A saída da ferramenta são instâncias do metamodelo *Database Schema* que podem ser usada para obter um novo esquema de banco de dados. A Figura 3.5 apresenta o processo que é dividido em três etapas.

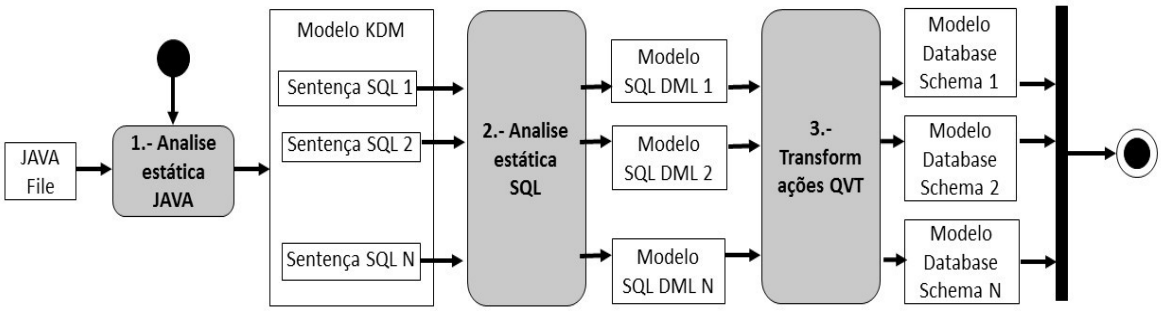

Figura 3.5 – Visão Geral da Contextualização de Dados adaptado de (PÉREZ-CASTILLO et al., 2009)

Na primeira etapa denominada de análise estática JAVA, obtém-se um Modelo KDM estendido a partir do código fonte. Na segunda etapa denominada de *Análise estática SQL*, realiza-se uma análise estática de cada sentença *SQL* contida no modelo KDM gerando vários modelos *SQL DML* em conformidade com o metamodelo proprietário *SQL DML*. Por fim, na última etapa denominada de transformações *QVT*, o modelo *Database Schema* é obtido a partir dos modelos de sentenças *SQL* por meio de um conjunto de transformações *QVT*. Para apoiar a técnica de contextualização, os autores propõem uma ferramenta que é estruturada em três módulos correspondentes às três atividades da técnica.

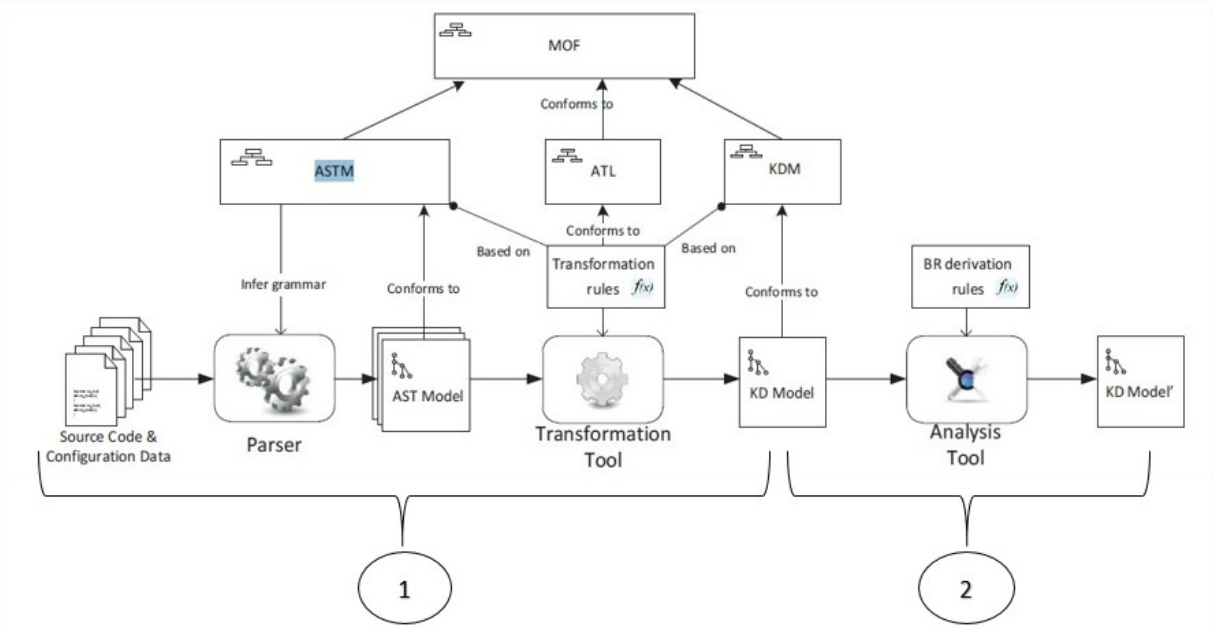

Figura 3.6 – Arquitetura da ferramentas adaptado de Vasilecas e Normantas (2011),

Um outro trabalho relacionado a este trabalho é o de Vasilecas e Normantas (2011), que propõem um processo para a obtenção de regras de negócios a partir de sistemas de informação legados. O resultado do processo é um conjunto descoberto de regras de negócio armazenadas no *modelo KD* (derivado do modelo KDM). A abordagem apresenta duas etapas como mostrada na Figura 3.6. Na primeira etapa denominada de preparação, os modelos *AST* são gerados a partir do código fonte do sistema. Depois, esses modelos *AST* são transformados para o modelo inicial KDM por meio da aplicação de um conjunto de regras de transformação ATL.

Na segunda etapa denominada de derivação de regras a partir do modelo KDM, os autores consideram o projeto de regras de negócios *GUIDE*, que classifica as regras de negócio nas seguintes quatro categorias: termos de negócios, fatos, restrições e derivações como mostrado na Figura 3.7. O resultado da aplicação da GUIDE é um conjunto de regras de negócio armazenado no modelo conceitual *KD*.

Kind of Business Rule (GUIDE)	Derivation from KDM model principles
Term	
Type	DataElement (CallableUnit, MethodUnit, StorableUnit, ItemUnit, MemberUnit, or ParameterUnit). Derivation must be considered according to their relationships with elements of UI, Data, and Event models.
Literal	ValueElement (Value or ValueList) that are assigned with DataElements.
Fact	
Base fact	ActionElement kind of Assign that relates a DataElement over relationship Writes with a DataElement or a ValueElement over relationship Reads.
Derived fact	Inference: An ActionElement kind of Assign that is successor of ControlFlow type of GuardedFlow (e.g Switch Statement), TrueFlow or FalseFlow (e.g. If Expression).
	Mathematic Calculation: An ActionElement kind of Assign that is successor of ControlFlow kind of Flow which consist of a set ActionElements kind of Actions related to primitive numerical datatypes (as defined in MicroKDM semantics)
Attribute	A relationship between abstracted KDM element ColumnSet which can be kind of DataSegment, RelationalView, RelationalTable or RecordFile, and is owner of a set of ItemUnit;
	a relationship ClassUnit and MemberUnit (in the case object-oriented design).
Participation	A relationship between CodeUnit and MemberUnit which is of ClassUnit datatype, or a relationship represented over KeyRelation element which indicates a relation between ReferenceKey (e.g. foreign key) and UniqueKey (e.g. primary key).
Generalization	A relationship represented over AbstractCodeRelationship kind of Extends which relates two Datatypes (from and to).
Derivation	
Mathematical calculation	A slice of ControlFlow with slicing criterion Fact, as its predefined in fact, derived by mathematical calculations.
Inference	A slice of ControlFlow with slicing criterion Fact, as its predefined in fact, derived by inference mechanism.
Action Assertion	
Condition	An ActionElement kind of Comparison which is ancessor for ControlFlow (TrueFlow and FalseFlow).
Integrity constraint	An ActionElement kind of Comparison which forces termination of ControlFlow (TrueFlow and FalseFlow).
Authorization	An ActionElement kind of Comparison which check for equality of two ComputationalObjects and forces termination of ControlFlow in false case.

Figura 3.7 – Resumo dos Principios de Derivacão de regras de Negócio (VASILECAS; NORMANTAS, 2011),

Na abordagem é interessante notar dois detalhes: um deles é que o modelo resultante *KD* não é gerado em conformidade com um metamodelo, isto é, o modelo *KD* é gerado a partir do modelo KDM e produto da aplicação da *GUIDE*. O Segundo detalhe é que para gerar o modelo *KD* não se utilizam regras de transformação, em vez disso se utiliza a ferramenta KDM Software Development Kit (SDK). Os autores não fornecem maiores detalhes do processo de obtenção do modelo de Regras de Negócio.

Assim como os trabalhos citados anteriormente, o trabalho de Canovas e Molina (2010) propõem uma ferramenta cujo objetivo é extrair métricas que permitem analisar o acoplamento de Triggers com Interfaces de usuário. A saída da ferramenta é um relatório de métricas. O processo de extração das métricas é feito em duas etapas. Na primeira etapa, gera-se o modelo KDM a partir do código procedural (Procedural Language (PL/SQL)). Na segunda etapa, os autores criam um mapeamento entre os elementos do metamodelo KDM e os elementos de um metamodelo de métricas proprietário. As transformações entre o KDM e o metamodelo de métricas foram desenvolvidas em RubyTL. Após a fase de geração da instância do metamodelo de métricas, um outro conjunto de transformações gera um arquivo de valores separados por virgulas (Comma-Separated Values (CSV)) para facilitar a visualização das métricas. A Figura 3.8 apresenta as etapas descritas.

O trabalho relacionado apresentado utiliza um metamodelo proprietário que restringe a reutilização da solução em projetos similares. Os autores não disponibilizam o metamodelo proprietário nem as regras de transformação para gerar o modelo de métricas a partir do metamodelo KDM.

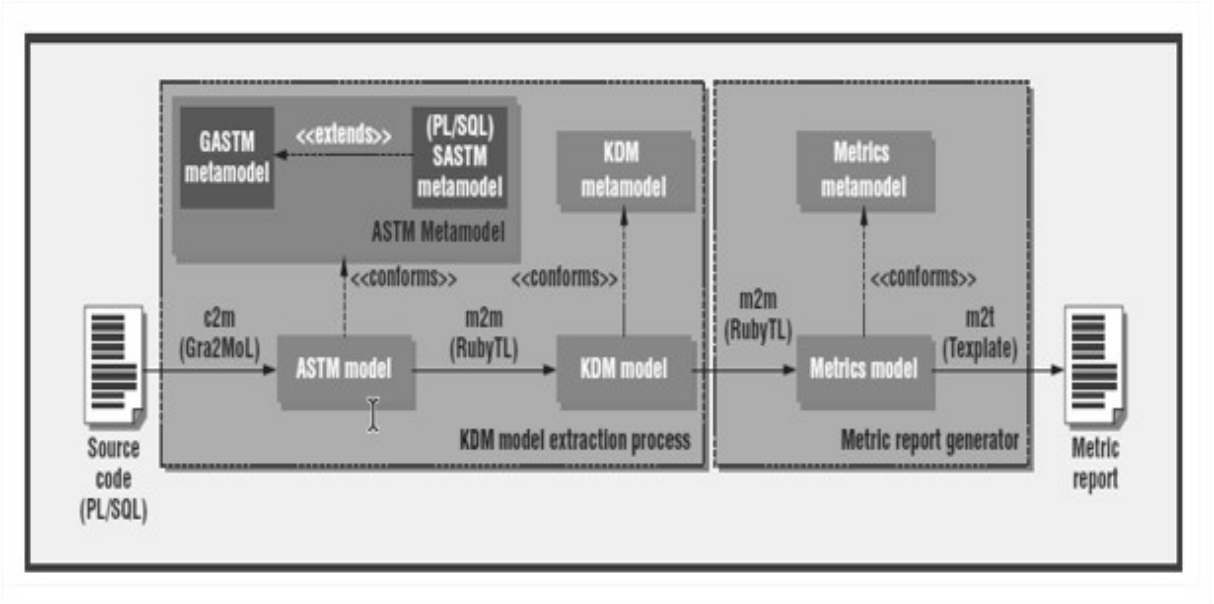

Figura 3.8 – Visão geral da ferramenta de extração de Métricas (PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2013)

Outro trabalho relacionado a este trabalho de pesquisa é o de Trias et al. (2014), que propõem uma ferramenta para sistematizar o processo de migração de Sistemas de gerenciamento

de conteúdo baseados em web (Content Management Systems (CMS)) para outros sistemas CMS. As atividades apresentadas estão enquadradas no estágio de engenharia reversa, concretamente nas tarefas para a geração do modelo KDM. Na Figura 3.9 é mostrado todo o processo de migração, mas o trabalho descrito corresponde com o quadrado de linhas pontilhadas. Para isto, os autores desenvolvem um ASTM_PHP DSL, uma linguagem de modelagem que permite definir modelos ASTM_PHP (extensão do metamodelo padrão ASTM). Segundo os autores o metamodelo ASTM-PHP posssui metaclasses suficientes para representar todos os detalhes da linguagem PHP. Em seguida, são implementados mapeamentos entre os elementos dos metamodelos ASTM_PHP e KDM e são desenvolvidas regras de transformação ATL baseado nesse mapeamento. A aplicação das regras de transformação permite a geração o modelo KDM a partir do modelo ASTM_PHP.

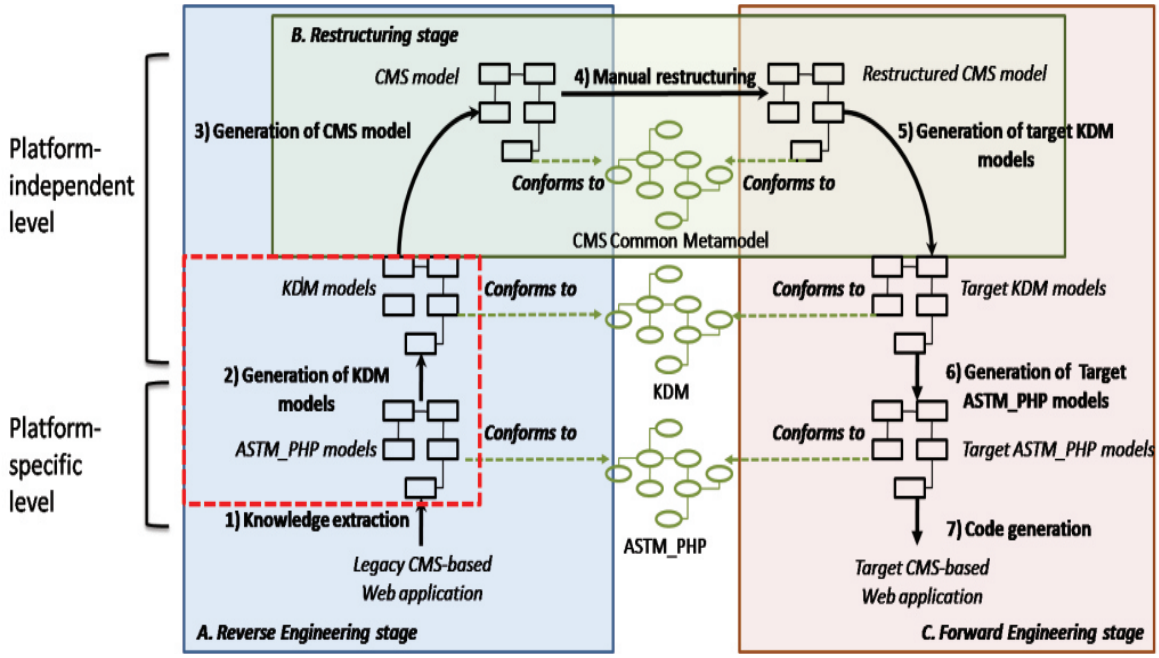

Figura 3.9 – Processo de Migração de Trias et al. (2014)

Os autores fornecem uma ferramenta que apoia o processo de migração, útil para casos de modernização que envolve sistemas implementados em PHP. Porém, não são fornecidos maiores detalhes do mapeamento entre os metamodelos KDM e ASTM_PHP, nem do desenvolvimento das regras de transformação.

Um outro trabalho relacionado a este trabalho é o de Wulf, Frey e Hasselbring (2012) que introduzem uma estratégia de transformação modular de três fases para a geracao do modelo KDM a partir de código-fonte C#. A primeira fase, chamada *Transformação de Tipo Interno*, tem como objetivo analisar o código-fonte C# do sistema fornecido e gerar o modelo AST com informações de *namespaces* e definições de tipo (por exemplo, classes, interfaces e estruturas) com seus modificadores e nomes correspondentes. A segunda fase, chamada *Declaração de Membro Interno e Definição de Métodos*, é responsável por transformar declarações de membros e definições de métodos, mas sem nenhum inicializador de membros e corpos de método.

Por fim, a terceira fase, chamada *Transformação de instrução*, é responsável pelo mapeamento de declarações C#, ou seja, os inicializadores dos membros e os corpos dos métodos são transformados. As três fases descritas abrangem sub-transformações distintas para os tipos, membros e métodos e declarações para obter como resultado o modelo KDM. A Figura 3.10 ilustra as partes e etapas de transformação envolvidas ao transformar um sistema C# em uma instância do KDM.

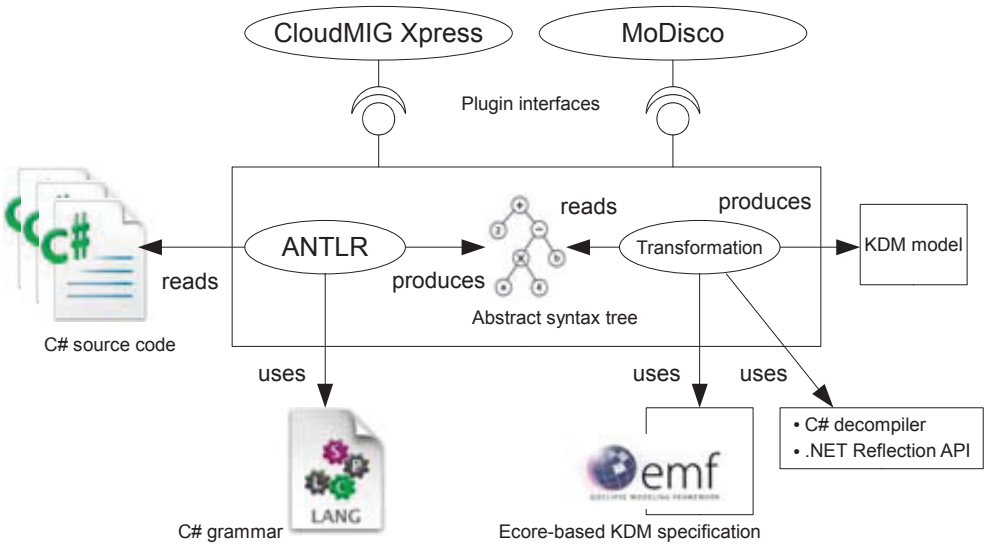

Figura 3.10 – Arquitetura de Transformação em três fases de Wulf, Frey e Hasselbring (2012)

Outros trabalhos relacionados a este trabalho são os de Pérez-Castillo, Guzmán e Piattini (2013), Pérez-Castillo, Guzmán e Piattini (2011), Pérez-Castillo et al. (2012), Pérez-Castillo et al. (2012), que propõem uma técnica denominada de arqueologia de processos de negócios que é apoiada pelo framework MARBLE, que é uma ferramenta de engenharia reversa baseada em KDM para recuperação de modelos de negócio. MARBLE é dividida em quatro níveis de abstração com três transformações entre eles como mostrado na Figura 3.11. As transformações entre os quatro níveis de abstração são: Transformação L0 para L1, consiste na transformação de código fonte para modelos Java; Transformação L1 para L2, consiste em um conjunto de transformações aplicados ao modelo Java para obter um modelo KDM; Transformação L2 para L3, permite a transformação do modelo KDM para o modelo de Processo de negócio (Business Process Model (BPM)). A transformação está baseada em um catálogo de padrões de correspondência entre os elementos dos metamodelos e implementada por meio de transformações QVT.

O interessante do framework MARBLE é que possui níveis de abstração definidos, transformações entre esses níveis e um catálogo de refatorações. O catálogo define 10 padrões de processos de negócio BPM e identifica as estruturas e metaclasses correspondentes no modelo KDM.

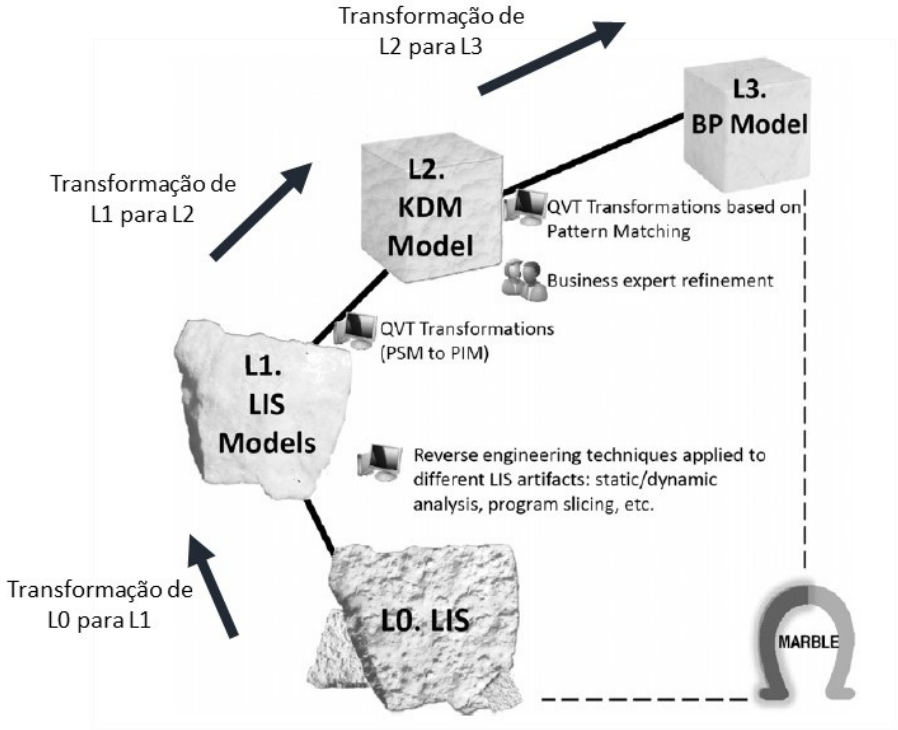

Figura 3.11 – Visão geral do MARBLE adaptado de Pérez-Castillo, Guzmán e Piattini (2011)

3.3 Considerações Finais

O objetivo neste capítulo foi mostrar um panorama de abordagens que se enquadram no contexto de modernização de sistemas, tanto focadas na engenharia reversa quanto nas outras etapas do processo de modernização. O intuito da revisão da literatura foi de identificar qualquer trabalho que possuísse máquinas de transformação que envolvessem o metamodelo KDM. Dentre esses trabalhos apresentam-se abordagens para geração de Modelo de diagramas de sequência UML (MARTINEZ; PEREIRA; FAVRE, 2014), Modelo SBVR (MORATALLA et al., 2012), Modelo de banco de dados *Database Schema* (PÉREZ-CASTILLO et al., 2009), Modelo de Métricas (CANOVAS; MOLINA, 2010), Modelo de Aplicações Cliente RIA (RODRÍGUEZ-ECHEVERRÍA et al., 2011), Modelo de Aplicações Móveis Android (PÉREZ-CASTILLO et al., 2013), Modelo KD derivado (VASILECAS; NORMANTAS, 2011) e Modelo de Negócio BPM (PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2013), (PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2011), (PÉREZ-CASTILLO et al., 2012), (PÉREZ-CASTILLO et al., 2012). Além disso, foram apresentadas duas abordagens para a geração do Modelo KDM a partir de código fonte PHP (TRIAS et al., 2014) e C# (WULF; FREY; HASSELBRING, 2012).

Todas as abordagens têm em comum dois aspectos: *i*) a criação de um mapeamento entre os elementos do metamodelo de origem KDM e do metamodelo alvo; e *ii*) o desenvolvimento de um conjunto de regras de transformação, baseado no mapeamento, encarregadas de transformar o modelo origem para o modelo alvo.

As principais diferenças encontradas nos trabalhos relacionados, quando comparados com

este trabalho de pesquisa são:

i) Os Metamodelos. No trabalho de geração do modelo de banco de dados (PÉREZ-CASTILLO et al., 2009), obtenção de modelo de métricas (CANOVAS; MOLINA, 2010) e derivação de regras de negócio (VASILECAS; NORMANTAS, 2011) os metamodelos utilizados são proprietários, isto é, particulares para o problema abordado impedindo a reutilização da solução em outros casos de modernização. Por outro lado, neste trabalho de pesquisa são utilizados metamodelos padrões e extensíveis como são o metamodelo KDM e Java, facilitando o uso da solução em diferentes cenários de modernização e permitindo a interoperabilidade entre ferramentas de modernização;

ii) Fase no processo de Modernização. A maior parte dos trabalhos relacionados se concentram na fase de engenharia reversa, enquanto a abordagem apresentada neste trabalho tem como foco o estágio de engenharia avante dos processos de modernização ADM;

iii) O tipo de metamodelo alvo. Nenhum dos trabalhos relacionados utiliza o metamodelo Java como PSM alvo da transformação;

iv) O tipo de transformação. Nos trabalhos relacionados as transformações M2M a partir do modelo KDM são do tipo exógena e vertical com aumento do nível de abstração, com excepção dos trabalhos de Moratalla et al. (2012) e Pérez-Castillo et al. (2013). O aumento no nível de abstração acontece porque esses trabalhos pertencem ao estágio de engenharia reversa. A abordagem proposta neste trabalho de pesquisa é focada em transformações exógenas e verticais com diminuição do nível de abstração.

Outro aspecto importante foi encontrado nos trabalhos do Framework GAFEMO (MORATALLA et al., 2012) e Modelo de Diagrama de Sequência (MARTINEZ; PEREIRA; FAVRE, 2014), os autores das abordagens consideram que o modelo KDM gerado pela ferramenta Modisco é incompleto. Por esse motivo, fazem um enriquecimento do modelo, isto é, desenvolvem regras de transformação adicionais para completar os dados faltantes no modelo KDM. Essas transformações de refinamento são descritas a alto nível e não são disponibilizadas para serem reutilizadas. No decorrer do trabalho de pesquisa enfrentamos dificuldades similares ao trabalhar com o metamodelo KDM, sendo necessário refinar o modelo para evitar a perda de informação. As soluções adotadas foram documentadas e disponibilizadas para serem reutilizadas e analisadas por interessados no modelo KDM.

Capítulo 4

ABORDAGEM PARA A CRIAÇÃO DE MOTORES DE TRANSFORMAÇÃO KDM2PSMS

4.1 Considerações Iniciais

Esse capítulo apresenta a abordagem para a criação de motores de transformação KDM para PSMs proposta neste trabalho. A abordagem é caraterizada por: *i*) ser iterativa e incremental permitindo criar uma regra de transformação KDM2PSM em cada iteração; *ii*) criar mapeamentos entre as instâncias dos modelos KDM e PSM gerados a partir de um *Requisito de transformação*; e *iii*) utilizar o *Mapeamento KDM-PSM* como principal fonte de informação para desenvolver as regras de transformação.

O capítulo detalha as fases e atividades que compõem a abordagem, exemplificando cada atividade para uma melhor compreensão. O capítulo está organizado da seguinte forma: na seção 4.2 é apresentada uma visão geral da abordagem, na seção 4.3 é descrita a primeira fase da abordagem e as atividades compreendidas nela, na seção 4.4 é apresentada a segunda fase da abordagem e as atividades envolvidas, na seção 4.5 é apresentada a terceira fase e as atividades compreendidas nela, na seção 4.6 é apresentada a quarta e ultima fase e as atividades envolvidas. Por fim, na seção 4.7 são apresentadas as considerações finais do capítulo.

4.2 Visão Geral da Abordagem

Esta seção apresenta a abordagem para criar motores de transformação do KDM para PSMs. O ponto principal da abordagem é dar suporte à elaboração de mapeamentos entre o metamodelo KDM e o metamodelo de algum PSM escolhido. Na abordagem proposta, isso é feito por meio da comparação entre instâncias desses metamodelos e o mapeamento resultante é usado para desenvolver as regras de transformação que constituem o motor de transformação.

Na Figura 4.1 é mostrada uma visão geral da abordagem proposta, que é constituída por quatro fases. A seta de retorno na parte inferior evidencia que a abordagem é iterativa e incremental, isto é, iterativa porque as fases podem se repetir várias vezes até que não exista outro

Requisito de transformação a ser resolvido e incremental porque toda vez que uma iteração é concluída, uma regra de transformação é criada ou regras existentes são evoluídas.

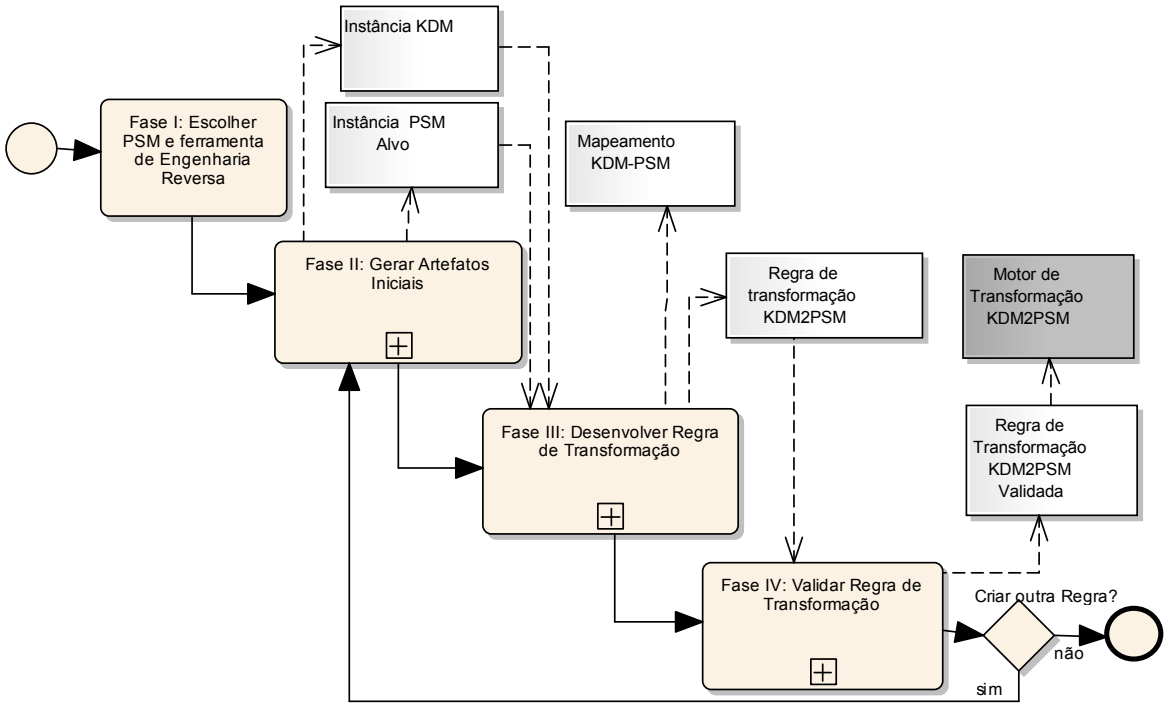

Figura 4.1 – Visão das Fases da Abordagem

A Fase I *Escolher PSM alvo e Ferramenta de Engenharia Reversa* tem como objetivos: definir qual PSM (Modelo Java, Modelo C #, Modelo Orientado a Serviços, etc) será usado como metamodelo alvo e escolher uma ferramenta de engenharia reversa que possa gerar a instância PSM (escolhido) a partir do código fonte.

A Fase II *Gerar Artefatos Iniciais* tem como objetivos: definir o Requisito de transformação, que neste trabalho é definido como uma estrutura representativa de código fonte; implementar um trecho de código fonte que represente esse requisito; e gerar instâncias dos metamodelos KDM e PSM, artefatos importantes utilizados no decorrer das atividades na abordagem. A Fase III *Desenvolver Regra de transformação* tem como objetivos: criar o mapeamento entre os metamodelos KDM - PSM e desenvolver uma regra de transformação baseada nesse mapeamento. A Fase IV *Validar Regra de transformação* tem o objetivo de testar a regra de transformação desenvolvida, verificando a completude do modelo java gerado. Consequentemente a junção de todas as regras desenvolvidas em cada iteração do processo constituem o motor de transformação KDM2PSM.

As próximas seções explicam com mais detalhes cada uma das fases mostradas na Figura 4.1.

4.3 Fase I: Escolher PSM alvo e Ferramenta de Engenharia Reversa

A primeira fase que é mostrada na Figura 4.2 tem objetivo duplo. O primeiro objetivo é definir qual PSM (Modelo Java, Modelo C #, Modelo Orientado a Serviços, etc) será usado como metamodelo alvo, já que as regras de transformação deverão gerar instâncias desse PSM escolhido. O metamodelo PSM deve possuir documentação completa e disponível de forma que possa ser consultada no decorrer das fases. O segundo objetivo é escolher uma ferramenta de engenharia reversa (ER) que gere instâncias desse metamodelo PSM a partir do código fonte. A ferramenta escolhida é utilizada na atividade *II*.2 para gerar a instância PSM alvo, que é um artefato importante para o desenvolvimento e validação da regra de transformação. A saída dessa fase é o metamodelo PSM e a ferramenta de engenharia reversa escolhidos.

Para exemplificar a abordagem, adotamos como PSM o metamodelo Java, que possui documentação completa e disponível. Além disso, escolhemos a ferramenta de engenharia reversa *MoDisco*. Modisco fornece descobridores chamados de *discover* que permite gerar a instância do metamodelo Java a partir do código fonte, assim como gerar a instância do metamodelo KDM.

4.4 Fase II: Gerar Artefatos Iniciais

Esta primeira fase tem como objetivo gerar os artefatos inicias para o correto desenvolvimento da abordagem. Para isto, o engenheiro de modernização inicia com a definição dos requisitos de transformação e com a implementação ou obtenção de um trecho de código que represente esse requisito. Neste trabalho se define um requisito de transformação como estruturas representativas de código fonte. Finalmente, o engenheiro de modernização deve gerar as instâncias KDM e PSM a partir do trecho de código fonte. Essas instâncias são peças importantes para o desenvolvimento da abordagem. A fase II possui três atividades como mostrado na Figura 4.2, que são descritas em detalhe a seguir.

Atividade II.1: Definir Requisito de Transformação. O objetivo dessa atividade é definir o *Requisito de transformação*, que é definido como uma estrutura de código fonte que uma regra de transformação a ser desenvolvida deverá gerar. Por exemplo, se o *Requisito de transformação* fosse a estrutura *while*, a regra de transformação resultante desta iteração deverá reconhecer como é representada essa estrutura no metamodelo do KDM e deverá ser capaz de gerar essa mesma estrutura no PSM de destino. Esta atividade depende da finalidade da geração do código. Às vezes, será necessário definir estruturas de códigos fonte completos que levem em conta os corpos, tipos e relacionamentos do método. Em outros casos, pode ser necessário apenas gerar apenas assinaturas de métodos, classes, atributos e parâmetros.

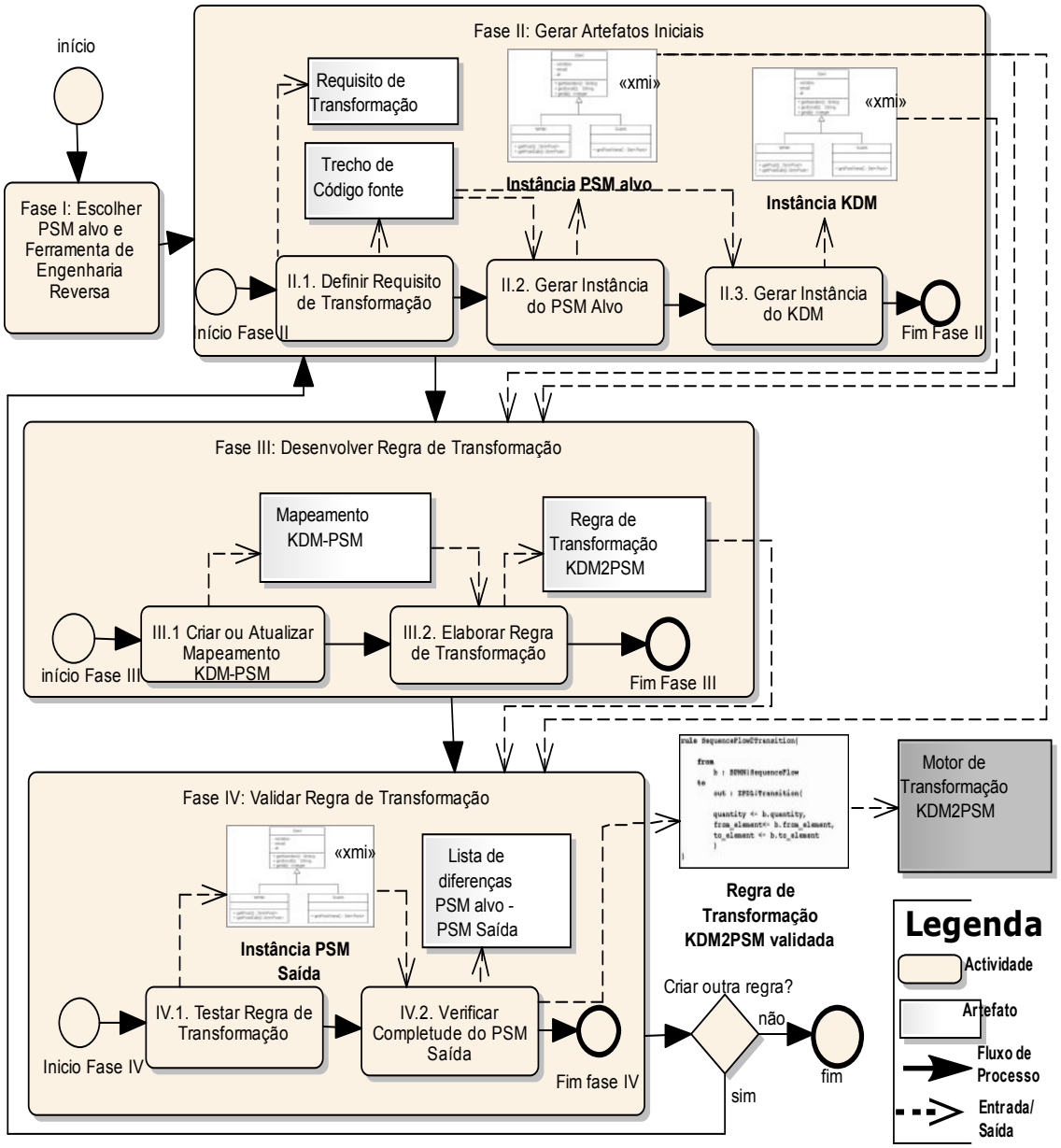

Figura 4.2 – Visão das Fases e Atividades da Abordagem

Depois de escolher a estrutura, o engenheiro de modernização deve implementar ou obter um trecho de código que represente a estrutura escolhida. A saída da atividade é o requisito de transformação definido e um trecho de código fonte que representa esse requisito e que é implementado ou obtido pelo engenheiro de modernização.

Para a exemplificação desta atividade, define-se como *Requisito de Transformação* a estrutura *if-then-else* e o trecho de código apresentado no Código 4.1 que representa esse requisito e que será utilizado no decorrer das atividades.

Atividade II.2: Gerar Instância do PSM Alvo. Esta atividade tem como objetivo a geração de uma instância PSM a partir de código fonte implementado na atividade anterior. Essa instância é um artefato importante por duas razões. Em primeiro lugar, na atividade *III*.1, ajuda

```
1 package dc;
2
3 class ifDemo {
4
5       public static void main() {
6
7           int testscore = 76;
8           boolean approved;
9
10          if (testscore >= 90) {
11            approved = true;
12          } else {
13            approved = false;
14          }
```

Código 4.1 – Código Fonte da Estrutura *If-Then-Else*

no desenvolvimento do mapeamento entre os metamodelos. Isto é, quando comparado com o modelo KDM permite a identificação das metaclasses equivalentes. Em segundo lugar, na atividade *IV*.2, o artefato é utilizado como *alvo* para verificar a integridade do PSM saída, isto é, o modelo PSM resultante da execução da regra de transformação (a ser desenvolvida) será comparado com esse PSM alvo para identificar os elementos diferentes no modelo. Note-se que essa instância do PSM serve como oráculo, pois o ideal é que a regra de transformação fosse capaz de gerar uma instância totalmente equivalente a essa que foi gerada aqui.

Para gerar a instância PSM alvo, o engenheiro de modernização precisa usar a ferramenta de engenharia reversa escolhida na Fase I e fornecer como entrada o código fonte desenvolvido na atividade anterior. O resultado dessa atividade é a instância do PSM que será chamada a partir de agora como *PSM Alvo*.

```
1 <?xml version="1.0" encoding="ASCII"?>
2 <java:Model ... name="CaseControlFlowStatementsIf">
3   <ownedElements name="dc">
4     <ownedElements xsi:type="java:ClassDeclaration" ... name="ifDemo">
5   ...
6    (A) <statements xsi:type="java:IfStatement" ...>
7      (B) <expression xsi:type="java:InfixExpression" ... operator=">=">
8          ...
9         </expression>
10     (C) <thenStatement xsi:type="java:Block" ...>
11         ...
12        </thenStatement>
13     (D) <elseStatement xsi:type="java:Block" ...>
14         ...
15        </elseStatement>
16     </statements>
17 ...
18 </java:Model>
```

Código 4.2 – Parte do Modelo PSM Alvo

Para a exemplificação desta atividade, a instância Java Alvo é gerada a partir do trecho de Código 4.1, que representa a estrutura *if-then-else*. Para isto, utiliza-se o suporte da ferramenta MoDisco e a opção *discover Java Project*. O Código 4.2 apresenta parte do Modelo Java ge-

rado. Nele, observa-se que a linha marcada com a letra *'A'* inicia a estrutura *if-then-else* que é representada com a metaclasse *IfStatement*. Depois, na letra *'B'*, a *expressão booleana* da estrutura *if-then-else* é representada com a metaclasse *InfixExpression*. Por fim, na letra *'C'* e *'D'*, a instrução *then* e *else* da estrutura *if-then-else* são representadas com a metaclasse *Block*.

Atividade II.3: Gerar Instância do KDM. O objetivo nesta atividade é a geração da instância do KDM a partir do trecho de código fonte desenvolvido. Essa instância é importante para esta abordagem porque apoia a identificação das metaclasses equivalentes. Como foi dito na atividade anterior, a instância KDM e instância PSM são os protagonistas na atividade *III*.1, encarregada da elaboração do mapeamento. Além disso, o modelo KDM é a entrada da atividade *IV*.1, fornecendo a informação para executar a regra de transformação desenvolvida.

Para a geração da instância do KDM, o engenheiro de modernização precisa usar uma ferramenta de modernização que gere a instância do KDM (PIM) a partir do código fonte. Na literatura, existem algumas ferramentas que podem automatizar este processo, tais como: Bruneliere et al (BRUNELIERE et al., 2010) e a ferramenta MoDisco para gerar o modelo KDM a partir do código fonte Java; Feliu Trias e outros (TRIAS et al., 2014), que fornecem uma ferramenta para gerar o modelo KDM a partir do código fonte do PHP; Christian Wulf et al (WULF; FREY; HASSELBRING, 2012), que apresentam uma abordagem para transformar programas C# em KDM e o software comercial BLUAGE (BARBIER et al., 2011) que pode transformar o código Cobol em KDM.

```
1 <?xml version="1.0" encoding="ASCII"?>
2 <kdm:Segment xmi:version="2.0" ...>
3 <model xsi:type="code:CodeModel" name="Case_ControlFlowStatements_If">
4   <codeElement xsi:type="code:Package" name="dc">
5     ...
6  (A)<codeElement xsi:type="action:ActionElement" name="if" kind="if">
7  (B)  <codeElement xsi:type="action:ActionElement" name="GREATER_EQUALS"
       kind="infix expression">
8        ...
9        </codeElement>
10 (C)   <codeElement xsi:type="action:BlockUnit">
11       ...
12       </codeElement>
13 (D)   <codeElement xsi:type="action:BlockUnit">
14       ...
15      </codeElement>
16    </codeElement>
17 ...
18 </kdm:Segment>
```

Código 4.3 – Parte do Modelo KDM

Para a exemplificação desta atividade, se utiliza também a ferramenta MoDisco e a opção *discover KDM* para gerar a instância do KDM a partir do trecho de código fonte *if-then-else*. O Código 4.3 mostra parte da instância KDM e com suporte da documentação se pode identificar os elementos no modelo. Na letra *'A'*, identifica-se o início da estrutura *if* que é representada com a metaclasse *ActionElement* com o atributo *kind* igual a *if*. Depois, na letra *'B'*, a *Expressão*

booleana é representada com a metaclasse *ActionElement* com o atributo *kind* igual a *infix expression*. Por fim, na letra *'C'* e *'D'*, a instrução *then* e a instrução *else* são representadas com a metaclasse *BlockUnit* na instância KDM. Note-se, que a instância KDM não possui uma metaclasse especifica para a estrutura *if-then-else*, utiliza o atributo kind igual a *if* para fazer a diferenciação. Além disso, os blocos de instruções *then* e *else* não apresentam algum elemento diferenciador pois eles são representados pela mesma metaclasse *BlockUnit*.

A Fase II descrita nesta subseção permite a geração dos artefatos inicias que são peças importantes para o desenvolvimento das próximas fases da abordagem. Os artefatos gerados são: a definição do *Requisito de transformação*, a implementação de um trecho de Código que implementa o Requisito definido, a instância KDM e a instância PSM alvo.

4.5 Fase III: Desenvolver Regra de Transformação

A Fase III, considerada a fase principal da abordagem tem como objetivo criar o mapeamento dos metamodelos KDM e PSM, assim como o desenvolvimento da regra responsável por transformar a instância do KDM que representa a estrutura de código fonte escolhida em uma instância do PSM representando a mesma estrutura. Assim, o engenheiro de Modernização cria o mapeamento entre os elementos (metaclasses e atributos) dos metamodelos, registrando cada equivalência no artefato *Mapeamento KDM2PSM*. Depois, baseado nesse conhecimento desenvolve a regra de transformação KDM2PSM. A fase possui duas atividades como mostrado na Figura 4.2 e as atividades são descritas com maior detalhe a seguir.

Atividade III.1: Criar ou Atualizar Mapeamento KDM-PSM. Esta atividade tem como objetivo estabelecer a equivalência entre as metaclasses dos metamodelos KDM e PSM que representam o *Requisito de Transformação*. O mapeamento é considerado a principal fonte de informação para o desenvolvimento da regra de transformação (seguinte atividade).

Para elaborar o mapeamento, inicialmente, o engenheiro de modernização deve procurar informação dos metamodelos como: regras de transformação PSM2KDM no estágio de engenharia reversa, documentação, artigos científicos, etc. que ajudem a entender os metamodelos e a transformação de forma mais profunda. O engenheiro de modernização deve revisá-lo concentrando-se nas metaclasses. Esse primeiro passo ajuda a incrementar o conhecimentos das metaclasses que fornecerá indícios do mapeamento que deve ser elaborado.

Seguidamente, ele deve realizar uma análise comparativa entre a instância do KDM, saída da atividade *II*.3, e a instância do PSM alvo, saída da atividade *II*.2. Essa comparação entre as instâncias permite identificar com mais precisão as metaclasses e os atributos equivalentes utilizados para representar a estrutura de código fonte escolhida.

Para a realização da análise, deve-se procurar nos modelos (arquivos XMIs) e identificar a metaclasse KDM e PSM utilizados para representar a mesma estrutura. Recomenda-se procurar pelas características atribuídas na implementação do trecho de código, por exemplo: nome e valores iniciais. Isto é, se o *Requisito de Transformação* fosse gerar um *Método* e no trecho

de código desenvolvido o método é chamado de *Soma*, então se deve procurar (nos arquivos XMI dos modelos) as metaclasses com o atributo nome igual a *Soma*, localizando assim as metaclasses PSM e KDM equivalentes.

Depois, se deve identificar os atributos presentes nas metaclasses descobertas. Isto é, gerar uma lista de atributos da metaclasse KDM (metaclase origem) e uma lista de atributos da metaclasse PSM (metaclasse destino). Após a identificação e sempre com suporte da documentação, o engenheiro de modernização deve determinar a equivalência semântica entre os atributos da lista. Entenda-se como equivalência semântica à dentificação dos atributos que representam o mesmo tipo de informação nos dois metamodelos, estabelecendo assim a correspondência entre eles. Nessa atividade é importante recopilar características dos atributo identificados como: *kind*, *type* e multiplicidade.

Por fim, o Engenheiro de Modernização deve identificar os atributos do metamodelo PSM que não apresentam uma correspondência com algum elemento do modelo KDM e vice-versa. Devido ao diferente nível de abstração o modelo PSM pode requerer informação mais detalhada da que é fornecida pelo KDM. Assim como, o modelo KDM pode conter informação redundante que não é necessária para o modelo PSM. O resultado dessa atividade é o artefato chamado *Mapeamento KDM-PSM*, que registra a equivalência entre os elementos PSM-KDM para a estrutura de código fonte em análise. Esse artefato é ativamente consultado e atualizado em cada iteração.

Para a exemplificação desta atividade, se compara o modelo Java Alvo apresentado no código 4.2, gerado no passo *II*.2, com a instância KDM mostrada no código 4.3, gerada no passo *II*.3. A análise entre as instâncias é feita por meio de letras e tem como resultado o mapeamento apresentado na Tabela 4.1. Para isto, primeiramente, procuramos a metaclasse que inicia a estrutura nos modelos, que de fato seriam as metaclasses de entrada e saída da regra de transformação a ser desenvolvida. Na instância KDM, observa-se na letra *A*, a metaclasse origem chamada *ActionElement* com *kind* igual a *if*. Da mesma forma, na instância PSM, observa-se na letra *A*, a metaclasse saída chamada *IfStatement*. Essa identificação permite estabelecer a primeira equivalência entre as metaclasses.

Tabela 4.1 – Mapeamento para a Estrutura If-Then-Else

Modelo KDM			**Modelo JAVA**
Pacote	**Metaclasse**	**Condição**	**Metaclasse**
Action	ActionElement	kind = 'if' kind = 'infix expression'	IfStatement InfixExpression
Action	BlockUnit		Block

Seguidamente, se analisam os atributos da metaclasse *ActionElement* (modelo KDM) e *IfStatement* (modelo Java). Na instância do KDM, observa-se na letra *B*, a metaclasse *ActionElement* com *kind* igual a *Infix Expression*. Da mesma forma, na instância PSM, observa-se na letra *B*, a metaclasse *InfixExpression*. Como resultado desta comparativa e com suporte da documentação dos metamodelos se pode estabelecer a equivalência entre a metaclasse *ActionElement*

com *kind* igual a *Infix Expression* e a metaclasse *InfixExpression*. Na letra *C*, na instância do modelo KDM, observa-se a metaclasse *BlockUnit* e na instância do modelo Java, observa-se a metaclasse *Block*. Com essa outra identificação pode-se estabelecer a equivalência entre as metaclasses *BlockUnit* e *Block*. Por fim, na letra *D*, a equivalência estabelecida na letra *C* é reiterada. A saída desta atividade é a criação do mapeamento entre as metaclasses dos modelos KDM e Java para a estrutura *if-then-else*, que é registradado no artefato *Mapeamento KDM-Java*. Note-se, na Tabela 4.1 a coluna *Pacote* se refere ao pacote do metamodelo KDM ao qual a Metaclasse analisada pertence. O artefato completo *Mapeamento KDM-Java* pode ser revisado no Apêndice A.

Atividade III.2: Elaborar Regra de Transformação. O objetivo desta atividade é desenvolver a regra de transformação para transformar a instância do KDM em uma instância do PSM preservando a estrutura de código fonte escolhida. O engenheiro de modernização deve apoiar-se principalmente no artefato *Mapeamento KDM-PSM*, na documentação dos metamodelos e na documentação da ferramenta de transformação que utilizará para desenvolver a regra.

Primeiro, ele deve estabelecer a metaclasse KDM de origem e a metaclasse PSM de destino, incluindo a condição ou filtro na metaclasse de origem para delimitar o conjunto que vai ser transformado. A identificação da condição ou filtro é importante porque no modelo KDM se observa o uso do atributo *Kind* de tipo *String* para distinguir dentre os diferentes tipos de metaclasses, isto é, o metamodelo não apresenta uma metaclasse diferente para cada tipo.

Após estabelecer a origem, destino e condições seguindo a sintaxe da linguagem de transformação escolhida, o engenheiro de Modernização deve continuar com o desenvolvimento do corpo da regra. Para isto, deve colocar cada atributo identificado da metaclasse PSM atribuindo o elemento equivalente da metaclasse KDM, de acordo com o mapeamento registrado no artefato *Mapeamento KDM-PSM*.

Na atividade de desenvolvimento do corpo das regras pode-se identificar os seguintes tipos de atribuições:

- Atribuição de constantes: quando um elemento da metaclasse PSM é atribuído com um valor constante, significa que o atributo é particular da metaclasse PSM, isto é, não apresenta um elemento equivalente na metaclasse KDM.

- Atribuição de metaclasse: quando o elemento da metaclasse PSM é atribuído com um uma metaclasse KDM, significa que existe uma correspondência direta de equivalência entre as duas metaclasses do modelo PSM e KDM.

- Atribuição com o valor retornado por uma função definida pelo usuário. A utilização de funções (helpers) permite a reutilização de informação, calcular algum valor ou principalmente para fazer conversões entre diferentes tipos de elementos.

Finalmente, a implementação de funções, o engenheiro de modernização deve implementar funções (helpers na linguagem ATL) para: (i) completar informação faltante que não pode

ser obtida a partir do modelo KDM de origem. Isto é, o modelo PSM precisa de informação detalhada como consequência do diferente nível de abstração; (ii) reutilização da informação, as funções permitem reutilizar o código desenvolvido em diferentes partes do programa; e (iii) realizar conversões entre os tipos de elementos, atividade comum em transformações verticais como a realizada nesta proposta de pesquisa.

```
1  rule TransformActionElementToIfStatement {
2   from source: KDM!ActionElement (source.kind='if')
3   to   target: JAVA!IfStatement (
4   originalCompilationUnit<-thisModule.getOriginalCompilatinUnit(source)
5   ,expression    <-source.codeElement
6   ,thenStatement<-source.codeElement
7                  -> select(e | e.oclIsTypeOf(KDM!BlockUnit))->first()
8   ,elseStatement<-if ( (source.codeElement
9                  -> select(e | e.oclIsTypeOf(KDM!BlockUnit))).size()>1)
      then
10                        source.codeElement
11                 -> select(e | e.oclIsTypeOf(KDM!BlockUnit))->last()
12                 else
13                   Sequence{}
```

Código 4.4 – Regra de transformação: *TransformActionElementToIfStatement*

Para a exemplificação desta atividade, desenvolve-se a regra de transformação para a estrutura *if-then-else* baseado no mapeamento feito na atividade anterior, com apoio da documentação dos metamodelos e com suporte da ferramenta de transformação ATL.

O código 4.4 mostra a regra de transformação *TransformActionElementToIfStatement* criada. Na linha 2, se estabelece a metaclasse origem da transformação, a metaclasse *ActionElement* com o atributo *kind* igual a *if* do metamodelo KDM. Na linha 3, se estabelece a saída da transformação, a metaclasse *IfStatement* do metamodelo Java. Com as duas primeiras linhas, se garante que cada instância da metaclasse *ActionElement* com *kind* igual a *if* no modelo KDM de entrada terá uma instância da metaclasse *IfStatement* no modelo Java de saída. Na linha 4, o atributo *OriginalCompilationUnit* é atribuído com uma função. O *helper* criado (funções na linguagem ATL) extrai a localização logica da classe Java à qual o *IfStatement* pertence. Na linha 5, o atributo *Expression* (modelo Java) é atribuído com o elemento *CodeElement*. De acordo com a documentação do metamodelo Java, o atributo *Expression* pode receber um conjunto de tipos de elementos incluindo o *Infix Expression*, por esse motivo nenhuma funcão (helper) é utilizado nessa atribuição. Na linha 6, o atributo *ThenStatement* do tipo *Block* é atribuído com o primeiro *CodeElement* do tipo *BlockUnit*. Por fim, na linha 8, o atributo *elseStatement* do tipo *Block* é atribuído com o segundo atributo *CodeElement* do tipo *BlockUnit*. Note-se, que nas duas últimas atribuições (linhas 6 e 8) precisa-se respeitar a ordem de aparição dos elementos *CodeElement*, já que o modelo KDM não apresenta uma diferenciação que indique qual elemento pertence a *then* ou *else* na estrutura *if-then-else*.

A Fase III, descrita nesta subseção, permite a criação do artefato *Mapeamento KDM-PSM* e baseado nesse artefato é implementada a Regra de Transformação KDM para PSM. Os artefatos

de saída de esta fase são: O mapeamento KDM-PSM e a Regra de transformação KDM2PSM para o *Requisito de transformação* definido.

4.6 Fase IV: Validar Regra de Transformação

Esta última fase tem como objetivo testar a regra de transformação desenvolvida e validar a completitude do modelo Java resultado. A fase possui duas atividades como mostrado na Figura 4.2 e como descrita a seguir.

Atividade IV.1: Testar Regra de Transformação. O objetivo desta atividade é testar a regra de transformação desenvolvida na atividade anterior, verificando se está correta, isto é, se atinge o requisito de transformação para a qual foi criada. Para isto, o engenheiro de modernização deve utilizar o toolkit da ferramenta de transformação, configurando como entrada a instância do KDM, gerada na atividade *II*.3. Seguidamente, executa-se a regra de transformação. No caso não se produza nenhuma saída deve-se revisar a sintaxes da linguagem de transformação no programa e fazer as correções, para isto, se retorna à atividade *III*.2. O resultado desta atividade é a instância do PSM chamada de *PSM saída*.

Atividade IV.2: Verificar Completude do PSM Saída. O objetivo desta atividade é verificar a completude do PSM de saída gerado na última atividade. A completude do Modelo PSM de saída indica que a regra de transformação foi desenvolvida corretamente e que está cumprindo com o requisito de transformação estabelecido no início da abordagem. Para isto, o engenheiro de modernização deve realizar uma analises comparativa entre o modelo PSM alvo, resultado da atividade *II*.2, com o modelo PSM saída, resultado da atividade *IV*.1. Essa comparação, que pode ser realizada manualmente o com suporte de alguma ferramenta livre, deve responder duas perguntas: Quais são os elementos diferentes no entre os modelos PSM? Por que esses elementos são diferentes? Para responder a primeira pergunta, os elementos diferentes resultado da comparativa entre os modelos são registrados em um artefato chamado *Lista de diferenças*. Para a segunda questão, todos os itens na lista de diferenças devem ser analisados procurando a identificação do problema raiz. Algumas das razões podem ser: (i) Atributos da metaclasse PSM com atribuição errada; (ii) funções (Helpers) que não retornam os dados corretos; e (iii) Modelo de entrada KDM está incompleto. A analises feita ajuda a refinar a regra de transformação da atividade *III*.2. A saída desta atividade é o artefato chamado *Lista de diferenças* e a identificação dos problemas que os produzem.

A Fase IV, descrita nesta subseção, permite validar a Regra de Transformação, para isto executa a regra de transformação ATL criada gerando uma instância PSM chamado de Saída. Também, aqui é gerado o artefato lista de diferenças, quando comparado a instância PSM Alvo (Fase I) e a instância PSM saída.

4.7 Considerações Finais

Neste capítulo foi apresentado a abordagem para a criação de motores de transformação KDM2PSM desenvolvida neste trabalho. A abordagem é caracterizada por usar como principal fonte de conhecimento o artefato *Mapeamento KDM-PSM* para a criação das regras de transformação. As quatro fases da abordagem foram descritas em detalhe, assim como as atividades que conformam cada fase. Além disso, foi apresentado um exemplo para o desenvolvimento de uma regra de transformação KDM para JAVA, considerando como *requisito de transformação* a estrutura *if-then-else*.

A abordagem apresentada é iterativa e incremental, permitindo acrescentar uma nova regra de transformação KSM2PSM ao motor de transformação com cada iteração. Consequentemente, enquanto mais iterações sejam feitas, o motor de transformação poderá suportar casos de modernização com maior complexidade.

No Capítulo 5 é apresentado o motor de transformação RUTE-K2J que permite a transformação do modelo KDM para o modelo Java. Também são apresentados os artefatos gerados, assim como as dificuldades enfrentadas no decorrer do desenvolvimento do motor.

Capítulo 5

RUTE-K2J MOTOR DE TRANSFORMAÇÃO KDM2JAVA

5.1 Considerações Iniciais

Esse capítulo apresenta RUTE-K2J (Rule-based Transformarion Engine KDM2JAVA), um motor de transformação que contribui com os projetos de modernização que seguem o padrão ADM. RUTE-K2J fornece suporte no estágio de Engenharia Avante, especificamente na transformação exógena vertical do modelo KDM (PIM) para o modelo JAVA(PSM).

O capítulo está organizado da seguinte forma: na Seção 5.2 apresenta-se a descrição do motor de transformação, a tecnologia utilizada e dos artefatos gerados. Na Seção 5.3 é apresentada as dificuldades enfrentadas. Por fim, na Seção 5.4 são apresentadas as considerações finais.

5.2 Descrição do Motor RUTE-K2J

Na literatura, existem vários trabalhos de reengenharia que descrevem como transformar o modelo KDM em uma determinada tecnologia usando componentes dedicados. Por exemplo, do modelo KDM-Extend para o SQL-DML (PÉREZ-CASTILLO et al., 2009), do modelo KDM para o modelo KD (VASILECAS; NORMANTAS, 2011) ou do modelo KDM para o Modelo de Métricas (CANOVAS; MOLINA, 2010). Nesses trabalhos se observam transformações do modelo KDM para modelos que estão em conformidade com metamodelos proprietários, dificultando a reutilização em trabalhos similares.

Além disso, são poucos os trabalhos que concluem o processo de modernização usando o modelo KDM no estágio de engenharia avante. Por exemplo, a transformação feita do modelo KDM para o modelo ASTM (MORATALLA et al., 2012), do modelo KDM para o modelo Android (PÉREZ-CASTILLO et al., 2013) ou do modelo KDM para o modelo RIA-Extended MDWE (RODRÍGUEZ-ECHEVERRÍA et al., 2011). Esses trabalhos apresentam transformações no estágio de engenharia avante, mas não disponibilizam as regras de transformação desenvolvidas para reutilização, análise ou aprimoramento.

Há uma lacuna no processo de modernização ADM, especificamente no estágio de engenharia avante para transformar automaticamente um modelo KDM para um modelo PSM, que se encontre em conformidade com um metamodelo padrão, conhecido e disponível. Para fechar essa lacuna propõe-se *RUTE-K2J*, um motor de transformação que tem como objetivo permitir a transformação exógena e vertical do modelo KDM para o modelo padrão Java.

RUTE-K2J, motor de transformação desenvolvido como parte deste trabalho de pesquisa, apresenta as seguintes vantagens:

- Contribuir com o estágio de engenharia avante do processo de modernização ADM, fornecendo uma solução de transformação a partir do modelo KDM (PIM) para o modelo JAVA (PSM). No entanto, o modelo KDM de entrada gerado pelo Modisco suporta apenas o pacote *Code* e parte do pacote *Action*;
- Permitir ser utilizado em diferentes casos de uso de modernização. Além disso, permite o trabalho em conjunto com ferramentas M2T para gerar código fonte Java, possibilitando finalizar a ferradura de modernização;
- Utilizar o modelo Java J2SE5 (modelo saída da transformação) que é um reflexo da linguagem Java;
- Utilizar o metamodelo Java que é definido na metalinguagem *Ecore* amplamente usada, que permite a modificação, extensão e reutilização por ferramentas de Metamodelagem. O metamodelo está disponível na fonte do plug-in *org.eclipse.gmt.modisco.java*;
- Fornecer acesso ao código fonte para análise, reutilização e aprimoramento.

5.2.1 Tecnologias Envolvidas

RUTE-K2J foi desenvolvida utilizando as seguintes tecnologias: *i*) a linguagem de transformação e kit de ferramentas ATL, que fornece maneiras de produzir um conjunto de modelos de destino a partir de um conjunto de modelos de origem (ECLIPSE, 2005); *ii*) a norma OCL (Linguagem de Restrição de Objeto) para os tipos de dados e expressões declarativas (OMG, 2006); *iii*) Eclipse Modeling Framework (EMF) que é um framework de modelamento fornecido pelo Eclipse; *iv*) Modisco e os discoverers que permitem a geração do Modelo Java e KDM a partir do código fonte; e *v*) Discover-Advanse, projeto ATL baseado no discover-KDM do Modisco, que inclui correções feitas nas regras ATL para gerar um modelo KDM refinado, garantindo a preservação da informação.

5.2.2 Artefatos do Motor RUTE-K2J

A primeira versão da RUTE-K2J é composta por: 55 regras de transformação desenvolvidas na linguagem de transformação ATL, 3 delas abstratas; 28 Helpers, que podem ser vistos como o

equivalente aos métodos na linguagem tradicional; e 10 regras Lazy, que são regras de transformação que precisam ser explicitamente chamadas. Como resultado do processo de desenvolvimento, obtém-se os seguintes artefatos:

i) O artefato *Mapeamento KDM-PSM*: apresenta o mapeamento entre as metaclasses equivalentes dos metamodelo KDM e Java. Esse artefato é considerado como a principal fonte de *conhecimento* para o desenvolvimento das regras de transformação porque identifica a metaclasse KDM de entrada e a equivalente metaclasse Java de saída. Note-se, que a ferramenta gera 56 metaclasses das 126 do metamodelo Java disponibilizado pelo projeto Eclipse. A Tabela 5.1 mostra parte do artefato *Mapeamento KDM-PSM*, a primeira coluna refere-se ao pacote do modelo KDM à qual pertence a metaclasse, na segunda coluna é mostrada a metaclasse KDM, a terceira coluna refere-se a condição ou filtro usado para delimitar as metaclasses KDM de entrada e a quarta coluna refere-se à metaclasse Java equivalente. Na Tabela 5.1, observa-se que a metaclasse *ActionElement*, que pertence ao pacote *Action* do metamodelo KDM, pode ser transformado em várias metaclasses do metamodelo Java como são: *IfStatement*, *InfixExpression*, *SwitchStatement*, *WhileStatement*, etc. Por esse motivo se deve utilizar como filtro o atributo *Kind* (terceira coluna) para limitar o tipo de metaclasse KDM. Efetivamente, o artefato é de suma importância porque o modelo KDM possui muitos elementos onde a distinção é feita por o atributo chamado *kind*, sem valor em alguma enumeração de elementos. O artefato completo *o Mapeamento KDM-PSM* pode ser pode ser revisado no Apêndice A ou acessado no seguinte URL: <https://goo.gl/cwBM9S> .

iv) O artefato *Código fonte ATL*: o conjunto de regras de transformação desenvolvidas conformam o motor de transformação RUTE-K2J. Essas regras permitem a transformação de um modelo KDM para um modelo Java, preservando a informação durante o processo de transformação. As regras foram desenvolvidas seguindo a abordagem iterativa e incremental descrita no Capítulo 4 e com suporte do kit de ferramentas ATL. O Código-fonte ATL das 55 regras de transformação, 28 Helpers e 10 regras Lazy estão armazenado no repositório GitHub e podem ser acessado no seguinte URL: <https://github.com/Advanse-Lab/RUTE-K2J>

ii) O artefato *Inventário de Regras de Transformação KDM2JAVA*: apresenta o inventário das regras de transformação ATL desenvolvidas. A Tabela 5.2 mostra parte do artefato. Observa-se que o nome da regra evidencia a metaclasse de origem e destino da transformação. Por exemplo, a regra *TransformCodeModelToJavaModel* tem como objetivo transformar a metaclasse *CodeModel* do metamodelo KDM para a metaclasse *Model* no metamodelo Java. Essa é a regra principal para estruturar o modelo Java e articular as outras regras. O artefato completo com as 55 *Regras de transformação* pode ser revisado no Apêndice B.

iii) O artefato *Inventário de Helpers ATL*: no contexto ATL, os *Helpers* podem ser vistos como o equivalente aos métodos. Eles podem ser chamados pelas regras de transformação ou por outros *Helpers* desde diferentes pontos do código. Os *Helpers* foram utilizados neste trabalho com o propósito de fatorar código, calcular informações, converter tipos de dados e filtrar informações. A Tabela 5.3 mostra parte dos *Helpers* desenvolvidos. Observa-se que o *Helper*

getOrphanTypes tem como objetivo retornar uma sequência de elementos *datatypes* presentes no modelo KDM de origem. O artefato completo com os 28 *Helpers* pode ser acessado no Apêndice C.

iv) O artefato *Inventário de Lazy rules*: as regras de transformação do tipo *Lazy* são regras que precisam ser explicitamente chamadas por outras regras para ser executadas. No desen-

Tabela 5.1 – Parte do Mapeamento KDM-JAVA

Modelo KDM			**Modelo JAVA**
Pacote	**Metaclasse**	**Filtro**	**Metaclasse**
Code	CodeModel (principal) CodeModel (External)	- - - -	Model
Code	Package	-	Package
Code	LanguageUnit	-	PrimitiveType
Code	ClassUnit	name <> 'Anonymous type' name = 'Anonymous type'	ClassDeclaration AnonymousClassDeclaration
Code	TemplateUnit	getRealTypeClassUnit() = True getRealTypeInterfaceUnit() =True	ClassDeclaration InterfaceDeclaration
Code	MethodUnit	kind = constructor kind = method	ConstructorDeclaration MethodDeclaration
Code	StorableUnit	kind <> local kind = local	FieldDeclaration VariableDeclarationFragment
Source	InventoryModel	-	CompilationUnit
Action	BlockUnit	-	Block
Action	ActionElement	kind = 'if' kind = 'infix expression' kind = 'postfix expression' kind = 'break' kind = 'case' kind = 'switch' kind = 'while' kind = 'method invocation' kind = 'class instance creation' kind = 'return' kind = 'assignment' kind = 'field access' kind = 'parenthesized' kind = 'this' kind = 'array access' kind = 'array creation' kind = 'array initializer'	IfStatement InfixExpression PostfixExpression BreakStatement SwitchCase SwitchStatement WhileStatement MethodInvocation ClassInstanceCreation ReturnStatement Assignment FieldAccess ParenthesizedExpression ThisExpression ArrayAccess ArrayCreation ArrayInitializer
Action	CatchUnit	Quando herda de 'ExceptionUnit'	CatchClause
Action	TryUnit	Quando herda de 'ExceptionUnit'	TryStatement
Action	Writes		SingleVariableAccess
Action	Reads		SingleVariableAccess
Action	Addresses		SingleVariableAccess

Tabela 5.2 – Inventario parcial das Regras de Transformação

Nº	Nome da Regra	Objetivo
1	TransformCodeModelToJavaModel	Regra principal para estruturar o Modelo Java a partir do Modelo KDM
2	TransformPackageToJavaPackage	Transformar a metaclasse *Package* do metamodelo KDM para a metaclasse *Package* do metamodelo Java
3	TransformClassUnitToClassDeclaration	Transformar a metaclasses *ClassUnit* do metamodelo KDM para a metaclasse *ClassDeclaration* do metamodelo Java
4	MethodUnitToConstructorDeclaration	Transformar a metaclasses *MethodUnit* do tipo *constructor* do metamodelo KDM para a metaclasse *ConstructorDeclaration* do metamodelo Java
5	TransformMethodUnitToMethodDeclaration	Transformar a metaclasses *MethodUnit* do tipo *method* do metamodelo KDM para a metaclasses *ConstructorDeclaration* do metamodelo Java
6	TrasformStorableUnitToFieldDeclaration	Transformar a metaclassses *StorableUnit* do Modelo KDM para a metaclasse *FieldDeclaration* do Modelo Java
7	TransformActionElementToInfixExpression	Transformar a metaclasses *ActionElement* do tipo *infix expression* do Modelo KDM a para metaclasse *InfixExpression* do Modelo Java
8	TransformActionElementToPrefixExpression	Transformar a metaclasses *ActionElement* do tipo *prefix expression* do Modelo KDM para a metaclasse *PrefixExpression* do Modelo Java
9	TransformActionElementToIfStatement	Transformar a metaclasses *ActionElement* do tipo *if* do Modelo KDM para a metaclasse *IfStatement* do Modelo Java
10	TransformActionElementToMethodInvocation	Transformar a metaclasse *ActionElement* do tipo *method invocation* do Modelo KDM a para metaclasses *MethodInvocation* do Modelo Java

volvimento da ferramenta, essas regras foram utilizados em conjunto com os *Helpers* com o proposito de caracterizá-los. A Tabela 5.4 mostra todas as regras *Lazy*. Observe-se que a regra *setOrphanTypes* permite caracterizar cada atributo retornado pelo Helper *getOrphanTypes*. O artefato com as 10 *Lazy Rules* também pode ser acessado no seguinte URL:<https://goo.gl/Y45cff>.

v) O artefato *Diagrama de dependências*: com as 55 regras de transformação ATL, elaborou-se um diagrama de dependências que fornece uma visão de como as regras de transformação se engrenam. O diagrama permite identificar a sequência de regras que precisam ser executadas para chegar até uma regra em particular. Na Figura 5.1 é mostrado parte do Diagrama. Nele observa-se o seguinte: As regras *R06* e *R07* são regras abstratas; a regra *R01* permite estruturar o modelo Java e sempre será a primeira em executar-se; a regra *R05* permite a geração de pacotes e para que seja executada primeiro deve ser executada a regra R01; a regra *R15* permite a geração

Tabela 5.3 – Parcial Inventario de Helpers ATL

Nº	Helper Name	Purpose
1	getOrphanTypes	Obter uma sequência de elementos *datatypes* (tipo de dados primitivos) no modelo KDM.
2	getCompilationUnit	Obter o inventario dos artefatos físicos do sistema do modelo KDM
3	getParametersMethod	Obter os elementos do Modelo External no modelo KDM
4	getReturns	Obter os retornos de um Método no modelo KDM
5	getType	Converter o tipo de dado enviado por parâmetro para o tipo TypeAccess do modelo Java
6	checkElementoExternal	Verificar se o elemento pertence ao Modelo External no modelo KDM
7	getFragment	Obter a variavel *fragment* dos elementos *Field Declaration*, variáveis locais e inicializador da estrutura *ForStatement*
8	getImportsporClasses	Obter os elementos *imports* de uma Classe no modelo Java
9	getParametro	Obter as variáveis nos parâmetros dos métodos no modelo Java
10	getOrdemElementos	Obter a ordem dos elementos no corpo de um método no modelo KDM

Tabela 5.4 – Inventario de Lazy Rules

Nº	Lazy Rule Name	Purpose
1	setOrphanTypes	Definir os atributos da metaclasse *orphanTypes* do Modelo Java
2	setCompilationUnit	Definir os atributos da metaclasse *compilationUnits* do Modelo Java
3	SetParametros	Definir os atributos dos parâmetros dos métodos no Modelo Java
4	SetTypeParametrosRetorno	Definir o tipo de dado do parâmetro de retorno do Metodono modelo Java
5	CreateExtends	Criar o elemento *Extends* para a extensão das classes abstratas
6	CreateTypeAccessWriteRead	Definir o elemento *Type* com informação da metaclasse *Attribute*
7	CreateSingleVariableWriteRead	Definir o elemento *variable* com informação da metaclasse *Attribute*
8	CreateAnnotation	Criar *Annotations*(@Override) para a implementação de métodos da classe abstrata
9	CreateImplements	Criar *implements* para a implementação de Interfaces
10	CreateUsesType	Definir os atributos da metaclasse *TypeAccess*

dos métodos e para que seja executada primeiro deve ser executada a regra R01, R05 e R011; e as regras R50, R48, R49, R30, R47, R29 são as regras chamadas *Folhas* porque nenhuma regra depende da execução de elas, mas elas dependem da execução de um conjunto prévio de regras. O artefato completo pode ser acessado no seguinte URL:<https://goo.gl/gcJMGr>.

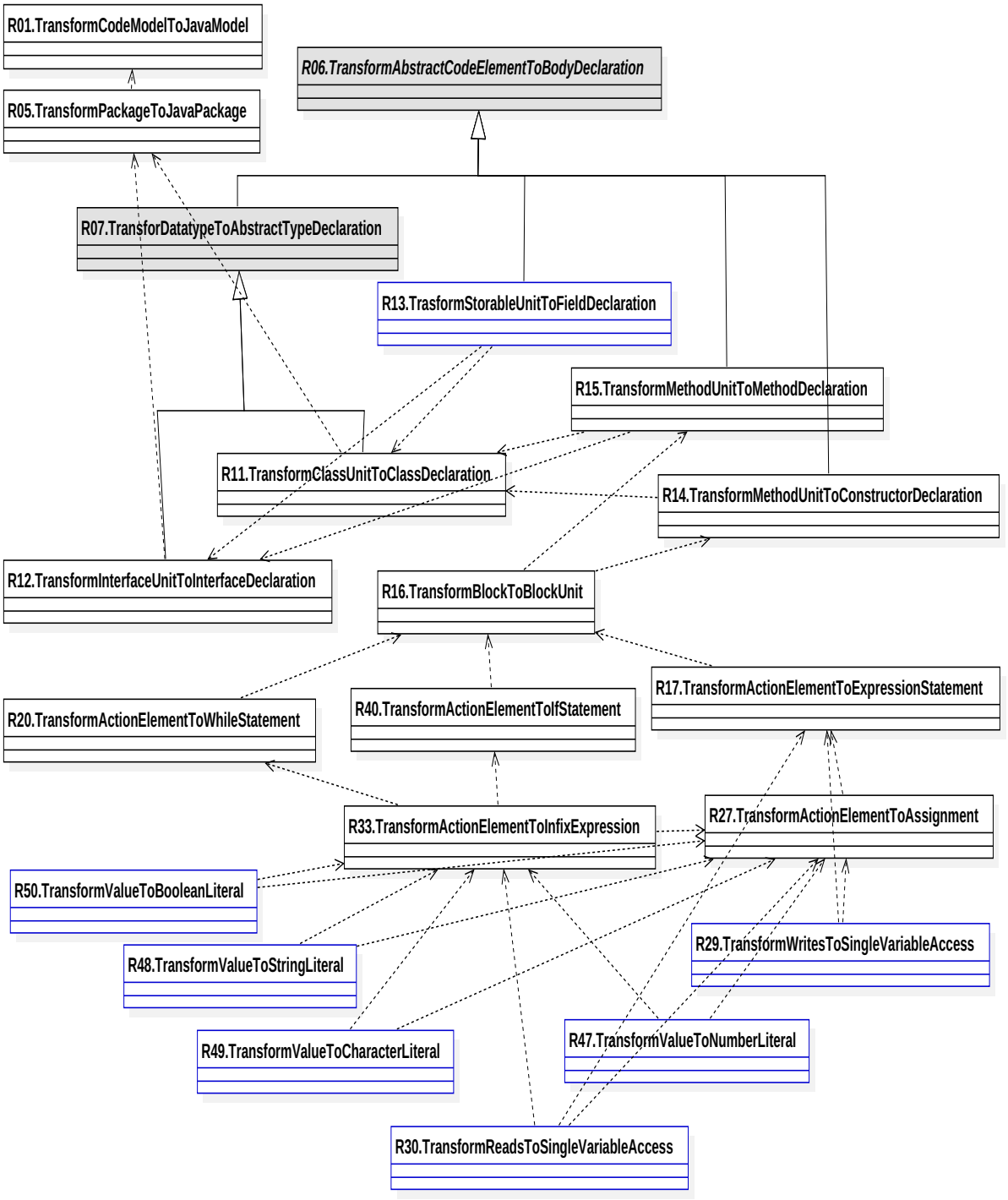

Figura 5.1 – Diagrama de dependências das Regras de Transformação ATL

5.3 Dificuldades Enfrentadas

Durante o desenvolvimento do motor de transformação *RUTE-K2J*, especialmente nas atividades de criação do mapeamento KDM-JAVA e no desenvolvimento das regras de transformação ATL, enfrentou-se as seguintes dificuldades que são descritas a seguir:

- Nível de abstração: O modelo Java, alvo da transformação, presenta um nível inferior de

abstração, isto é, possui metaclasses que precisam de mais dados da que é fornecida pelo modelo KDM. Por exemplo, a metaclasse *CompilationUnit* possui os siguentes atributos: *name*, *originalFilePath*, *types* e *imports*. Por outro lado, a metaclasse KDM equivalente chamada *InventoryItem* tem apenas os atributos *name* e *originalFilePath*. A informação faltante foi obtida com apoio de *Helpers ATL*, garantindo a geração de um modelo Java completo.

- Ordem dos elementos: O modelo KDM tem metaclasses com atributos do mesmo tipo que não possuem alguma característica que indique a ordem do elemento, isto é, não se conhece de antemão qual elemento deve ser atribuído primeiro ou último. Por exemplo, a metaclasse *ActionElement* pode ter zero ou muitos atributos *codeElements* do mesmo tipo sem nenhuma característica indicando a posição do elemento no modelo. Por outro lado, no modelo Java, as metaclasses apresentam atributos com nomes que indicam a ordem. Por exemplo, a metaclasse *InfixExpression* tem os atributos *RightOperand* e *leftOperand* que identificam a posição dos elementos que representam, facilitando o desenvolvimento das regras de transformação. Para o desenvolvimento da RUTE-K2J, considerou-se que a ordem dos elementos no modelo KDM estão de acordo com a ordem de aparição no modelo. Manter a ordem correta dos elementos garante preservar a coerência das expressões.

- External Model: O Modelo KDM esta conformado por 3 modelos, um deles é o modelo chamado *External* que armazena as referências das classes externas (classes importadas). No entanto, no modelo *Principal* existem expressões como declaração de variáveis locais com atribuição de valores iniciais que têm esses valores colocados no modelo *External*. Os diferentes tipos de elementos colocados nesse modelo dificulta a separação e o reconhecimento dos elementos. Neste trabalho, desenvolveu-se Helpers (funções) para realizar a distinção dos elementos contidos no modelo *External*.

- Tipos de Metaclasses: Um dos principais problemas enfrentados aqui foi, que o Modelo KDM não armazena os valores constantes em estruturas de dados tipo enumeração. Dificultando a identificação de todos os possíveis valores que pode adotar um atributo. Por exemplo, os valores da visibilidade da Classe (Private, Public e Protected) são obtidos da variável *value* do tipo *string*. Além disso, os tipos da metaclasses *ActionElement* como *For*, *If*, *Switch*, etc., são obtidos da variável *Kind* do tipo *String*. Todos esses valores foram identificados com apoio dos trechos de código fornecidos durante o desenvolvimento do motor de transformação.

- Estrutura de fluxo de controle *For*: O modelo KDM usa a metaclasse *ActionElement* do tipo *Variable Declaration* para a declaração de variáveis locais, assim como para a inicialização da variável de controle em uma estrutura *For*. No entanto, o modelo Java usa o *VariableDeclarationStatement* para variáveis locais e *VariableDeclarationExpression*

para a inicialização da variável de controle *For*. Ou seja, a metaclasse *Variable Declaration* do KDM pode-se transformar em duas metaclasses JAVA diferentes que são: *VariableDeclarationStatement* e *VariableDeclarationExpression*. A dificuldade aqui está na eleição da metaclasse Java correta porque a metaclasse KDM (*Variable Declaration*) não possui nenhuma característica diferenciadora que ajude na escolha. Para o trabalho de pesquisa, desenvolveu-se um *helper ATL* para identificar se a variável analisada pertence a um estrutura *For*.

- Estrutura de fluxo de controle *Switch*: No modelo KDM, a estrutura *Switch* não apresenta diferenciação entre o elemento *Case* e o elemento *Default Case*. Neste caso, se deduz que a metaclasse *Case* sem condições para ser executado é o elemento *Default Case*.

- Elementos aninhados: As estruturas que apresentam muitos elementos aninhados não são geradas por completo pela ferramenta Modisco. A documentação do modelo KDM não fornece exemplos claros desse tipo de estruturas, nem informação detalhada de como devem estar representados. Por exemplo, a sentença de código fonte *System.out.println (Variável)*.

No desenvolvimento da ferramenta RUTE-K2J, tem-se detectado informações faltantes no modelo de entrada KDM gerado pelo discover da ferramenta Modisco. Por esta razão, se colaborou com o projeto Modisco na revisão e modificação das regras de transformação que geram o modelo KDM, garantindo assim a geração de um modelo refinado que incorpora a informação faltante. Os bugs a seguir foram registrados e aceitados pelo projeto Modisco de Eclipse: *i*) *Bug 525331* corrige a atribuição do atributo *isAbstract* com o valor correto; *ii*) *Bug 526229* corrige a perda da variável *SinglevariableAcess* durante a transformação; *iii*) Bug 52623 corrige o valor de guarda no fluxo de controle *switch*, valor que se encontrava ausente.

Porém, as melhorias só serão disponibilizadas em uma próxima versão da ferramenta Modisco. Por isso, se utiliza uma versão do *discover* chamado de *DiscoverKDM-Advanse* que é baseado no Modisco e que inclui as correções feitas. O código ATL se encontra armazenado no repositório GitHub e pode ser acessado no seguinte URL: <https://github.com/Advanse-Lab/Discover-Advanse>;

5.4 Considerações Finais

Neste capítulo foi apresentado o motor de transformação RUTE-K2J, motor de transformação que fornece suporte para a transformação do modelo KDM para o modelo Java com o intuito de ajudar aos engenheiros de modernização a completar o processo de modernização ADM.

Além disso, foram descritos os artefatos resultantes do desenvolvimento que são: Inventário de Regras de transformação, Inventário de Helpers, Inventário de lazy rules, e o Diagrama de Dependência entre as regras de transformação. Também, foram descritas as dificuldades

enfrentadas na criação do mapeamento KDM-JAVA e no desenvolvimento das regras de transformação, assim como a solução adotada para cada situação. Por fim, foram descritas as contribuições feitas no projeto Modisco para completar a informação faltante que foi detetada no decorrer do desenvolvimento, dando origem ao descobridor *DiscoverKDM-Advanse* utilizado neste trabalho de pesquisa.

O código ATL da ferramenta RUTE-K2J é disponibilizado e pode ser acessado para analise, evolução e aprimoramento no seguinte URL: <https://github.com/Advanse-Lab/RUTE-K2J>.

Capítulo 6

AVALIAÇÃO DA FERRAMENTA RUTE-K2J

6.1 Considerações Iniciais

No Capítulo 4 é apresentada uma solução para apoiar o engenheiro de modernização a criar motores de transformação a partir do metamodelo KDM para qualquer PSM. Além disso, no Capítulo 5, é apresentada RUTE-K2J uma ferramenta de transformação desenvolvida seguindo as diretrizes da abordagem apresentada e que auxilia nas transformações de metamodelo KDM para modelos Java.

Neste capítulo apresenta-se a avaliação ao motor de Transformação RUTE-J2K, que visa verificar se as regras de transformação, que conformam o motor, cumprem com os objetivos de criação. Além disso, avaliar se a combinação das regras, de forma aleatória, resulta em transformações corretas.

O capítulo está organizado da seguinte forma: na Seção 6.2 é apresentado o desenvolvimento e definição da avaliação. Na Seção 6.3 é apresentada a Metodologia de definição dos Casos de Teste, que inclui a seleção da mostra, projeto de Caso de Teste e a implementação dos Casos de Teste. Na Seção 6.4 se apresenta a execução dos Casos de Teste. Na Seção 6.5 se apresenta a Análise dos Resultados da avaliação. Na Seção 6.6 é apresentada as Ameaças à validade. Por fim, na Seção 6.7 são apresentadas as considerações finais.

6.2 Desenvolvimento e Definição da Avaliação

Nesta seção é apresentada a avaliação ao motor de transformação RUTE-K2J que tem como objetivo principal levantar evidências da qualidade das regras de transformação, que compõem o motor, para que os engenheiros de modernização possam utilizá-lo em seus projetos de modernização.

Defini-se o escopo da avaliação, utilizando a organização proposta pelo modelo Meta/Pergunta/-Métrica (GQM) (WOHLIN et al., 2012) para fazer isso. De acordo com esse modelo de definição de metas, o escopo do estudo pode ser resumido da seguinte forma.

Analisar se todas as regras de transformação da RUTE-K2J funcionam como esperado
para fins de avaliação
com relação a corretude de todas as regras de transformação [1]
do ponto de vista do pesquisador
no contexto de engenheiro de software, ou seja, engenheiro transformando o modelo KDM para o Modelo Java.

6.3 Metodologia de Definição dos Casos de Teste

A ferramenta RUTE-K2J é composta por 52 regras de transformação padrão e 3 regras abstratas. Cada regra foi desenvolvida individualmente a partir de um requisito de transformação, isto é, com o objetivo específico de gerar uma estrutura de código-fonte, por exemplo: a estrutura *If*, *For*, criação de métodos, atributos de uma classe e atribuição de variáveis. Note-se, que a ferramenta RUTE-K2J gera 56 metaclasses das 126 do metamodelo Java disponibilizado pelo projeto Eclipse.

Neste contexto, para a avaliação da RUTE-K2J definimos como estratégia utilizar casos de testes conformados por programas Java, escolhidos segundo a seleção de amostra descrita no ponto 6.3.1 e o processo definido no ponto 6.3.2, com o intuito de validar se as regras satisfazem os objetivos de criação e se a combinação delas, de forma aleatória, resulta em transformações corretas. Além disso, o uso dos casos de testes permite garantir o exercício das 52 regras padrão pelo menos uma vez.

6.3.1 Seleção de Amostra

A análise se foca em projetos de código aberto para que o estudo possa ser facilmente replicado e ampliado. Dado que a amostra foi selecionada aleatoriamente a partir de repositórios de fontes abertas, tentou-se incluir uma ampla variedade de códigos-fonte que diferem em tamanho e complexidade. Portanto, se selecionou alguns repositórios Java ordenados por popularidade no GitHub. Os seguintes critérios foram estabelecidos na construção da amostra:

- *Projetos de código Aberto:* Selecionou-se aleatoriamente programas de código aberto hospedados em repositórios do GitHub [2]. Seguiu-se as diretrizes propostas por (KALLIAMVAKOU et al., 2014) durante a construção da amostra.
- *Java:* tem pelo menos 90% do repositório de código efetivamente escrito em Java.

[1] Aqui, uma regra de transformação está correta ao transformar com êxito um elemento de origem em um elemento esperado (alvo)
[2] <https://github.com>

6.3.2 Projetar Caso de Teste

A Figura 6.1 apresenta as 4 atividades para projetar os casos de teste e limitar o subconjunto de regras de transformação a serem testadas. Para isto, na Atividade 1, se define uma matriz com todas as regras de transformação e se identificam as dependências entre elas. Repare-se que das 55 regras de transformação são utilizadas as 52 regras padrão, não se consideram as 3 regras abstratas. Seguidamente, na Atividade 2, as dependências identificadas por cada regra são contabilizadas para selecionar as regras chamadas de *folha*, gerando assim a Matriz de Regras folhas. Neste trabalho, uma regra *folha* é aquela que presenta dependência de outras regras, mas nenhuma outra regra depende de ela. Na Atividade 3, a Matriz de Regras Folhas é priorizada. Finalmente, na Atividade 4, as Regras *folhas* são selecionadas e combinadas para escolher os dados de teste (programas Java), garantindo que todas as regras de transformação (52 regras) serão executadas pelo menos uma vez com o menor número de casos de teste.

Cada atividade do processo é descrita com maior detalhe a seguir:

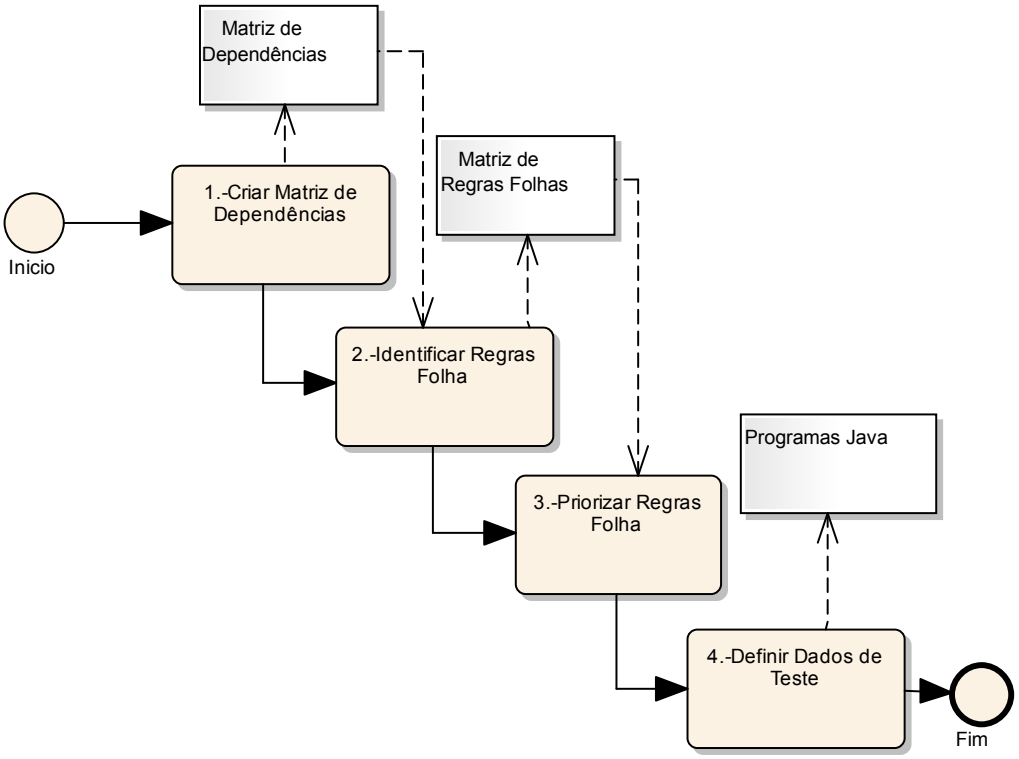

Figura 6.1 – Processo para Projetar Casos de Teste

1. *Criar Matriz de Dependências*: Esta atividade tem como objetivo elaborar uma matriz com as regras de transformação padrão, nela são identificadas os tipos de dependências: Obrigatória, Opcional e Indireta existentes entre as regras de transformação.

   Primeiro, marca-se a *Dependência obrigatória*, isto é, a identificação de regras responsáveis pela geração da estrutura básica do Modelo Java. Para isto, selecionamos cada

regra posicionada na vertical da matriz e marcamos com o símbolo *"*"* as dependências obrigatórias com as regras posicionadas na horizontal. Por exemplo, R01 é uma regra obrigatória porque ela é a responsável de gerar o esqueleto do modelo Java, sendo assim, todas as regras de transformação têm uma dependência obrigatória com essa regra.

Segundo, marca-se a *Dependência opcional*, isto é, a identificação de regras não obrigatórias. Na realidade, existem tantas combinações de regras de transformação quanto estruturas de código fonte para ser representados. Neste trabalho de pesquisa, as regras opcionais são identificadas baseadas nos trechos de código usados no processo de criação das regras de transformação. Para marcar as regras opcionais na matriz, selecionamos cada regra de transformação (na vertical) e marcamos com o símbolo *"**"* a dependência com alguma outra regra na horizontal. Por exemplo, a regra R13 encarregada de gerar atributos pode depender opcionalmente da regra R11 (geração de classes) ou da R12 (geração de interfaces) segundo o tipo de estrutura de código fonte que deve gerar.

Finalmente, marca-se a *Dependência indireta*, existem algumas regras de transformação que precisam ser combinadas com outras para construir uma estrutura completa. Por exemplo: Se a regra de transformação 'A' depende da execução da regra 'B', mas 'B' e 'C' conformam uma estrutura de código-fonte, então 'A' vai depender indiretamente da regra 'C'. Esse tipo de dependência é marcada com *ID-(Regra)*, onde *Regra* indica a regra em análise.

O resultado desta primeira atividade é a Matriz de Dependências.

2. Identificar Regras *Folha*: Esta atividade tem como objetivo a identificação das regras *folha* na matriz de dependências elaborada na atividade anterior. Para este trabalho, as regras *folha* são regras que não são requeridas por outras regras para completar sua execução. Isto é, nenhuma regra depende delas, no entanto, essas regras podem depender de várias outras regras. Essas regras ajudam no propósito de garantir que todas as regras de transformação sejam executadas pelo menos uma vez. Para identificá-las, deve-se contabilizar a quantidade de marcas (dependências) de cada coluna da matriz e selecionar as regras com valor igual a zero. O resultado desta atividade são as regras *folha* identificadas;

3. Priorizar Regras *Folha*: Gera-se uma nova matriz considerando as regras *folha* na vertical e mantendo as regras de transformação na horizontal. Depois, para a priorização se contabiliza as dependências de cada *folha* somando a quantidade de marcas (dependências) em cada linha na matriz. Seguidamente, considerando os valores obtidos, se priorizam as *folha* com maior valor colocando elas no topo da matriz. A saída desta atividade é a matriz de Regras *Folha* priorizada.

4. Definir Dados de Teste: Esta atividade tem como objetivo a seleção dos dados de teste representados pelos programa Java. Para a escolha dos dados de teste se adota como

critério a combinação de regras *folha* priorizadas, pois isto garantirá a execução de todas as regras de transformação no menor número de casos de Teste.

Então, seleciona-se um programa (código fonte) Java disponível, segundo o ponto 6.3.1, que tenha maior cobertura das regras *folha* combinadas, isto é, que execute as regras *folha* e a maior quantidade das regras das que ela depende. Repetindo esse exercício até garantir que todas as regras estão sendo cobertas pelos programas Java.

O resultado desta atividade é o programa Java por cada caso de teste, assim como o conjunto de regras de transformação a serem testadas.

6.3.3 Implementar Caso de Teste

Após projetar os Casos de Teste, implementamos cada uma das atividades descritas para obter como resultado os Casos de teste e o conjunto de regras de transformação a serem testadas.

1. Criar Matriz de Dependências: Para elaborar a matriz de dependências se devem colocar as 52 regras de transformação da ferramenta RUTE-K2J na ubicação horizontal e vertical (não são consideradas as regras abstratas). Depois, começando com a primeira regra de transformação posicionada na vertical se identificam as *dependências obrigatórias, direitas e indiretas* com respeito às regras colocadas na horizontal. Esse exercício se repete até completar as 52 regras padrão colocadas na vertical.

   A Tabela 6.1 apresenta parte da *Matriz de Dependências*, resultado da implementação desta atividade. Observa-se, a regra R30 (ler variável) tem dependência obrigatória com a regra R01 (gerar modelo Java), regra R05 (gerar pacotes), regra R11 (gerar classes), regra R15 (gerar métodos) e regra R16 (gerar bloco de ações) marcadas com o símbolo *"*"*. As regras mencionadas são as responsáveis pela criação da estrutura básica do modelo Java, por isso são executadas antes da regra (R30) em análise.

   Além disso, a mesma regra R30 depende opcionalmente da regra R14 (gerar construtores), regra R18 (gerar retorno dos método) ou da regra R21 (gerar estrutura de controle 'Switch'), que são marcadas com o símbolo *"**"*. A dependência opcional, como dito anteriormente, indica que a regra de transformação pode depender de uma ou de todas elas segundo o tipo de estrutura de código-fonte a gerar.

   Finalmente, a regra R30 que depende opcionalmente da regra R21 (gerar estrutura 'Swith') vai depender indiretamente da Regra R22 (gerar estrutura Break) e regra R23 (gerar estrutura Case) que são marcadas com *ID-R21*. Isto significa que as regras de transformação R21, R22 e R23 juntas compõem a estrutura *SwithStatement* e que se R30 depende de uma delas regras vai depender indiretamente das outras.

Tabela 6.1 – Matriz parcial das Regras de Transformação

Regra	R01	R05	R11	R13	R14	R15	R16	R18	R20	R21	R22	R23	...
R13 TrasformStorableUnitToFieldDeclaration	*	*	*		-	-	-	-	-	-	-	-	...
R29 TransformWritesToSingleVariableAccess	*	*	*	-	**	*	*	-	-	-	-	-	...
R30 TransformReadsToSingleVariableAccess	*	*	*	-	**	*	*	**	-	**	ID-R21	ID-R21	...
R31 TransformateAddressesSingleVariableAccess	*	*	*	-	**	*	*	-	-	-	-	-	...
R33 TransformActionElementToInfixExpression	*	*	*	-	**	*	*	**	**	-	-	-	...
R37 TransformActionElementToMethodInvocation	*	*	*	-	**	*	*	**	-	**	-	-	...
R38 TransformActionElementToForStatement	*	*	*	-	**	*	-	-	-	-	-	-	...
...	...	...	...	...	....	...	...	...	...	...	...	...	...
Occurrences	51	47	42	0	34	38	38	5	4	8	7	7	...

Tabela 6.2 – Matriz de regras *Folhas*

Regra	R01	R05	R11	R13	R14	R15	R16	R18	R20	R21	R22	R23	...	Total
R47 TransformValueToNumberLiteral	*	*	*	-	**	*	*	-	-	ID-R23	ID-R23	**	...	25
R30 TransformReadsToSingleVariableAccess	*	*	*	-	**	*	*	**	-	**	ID-R21	ID-R21	...	22
R48 TransformValueToStringLiteral	*	*	*	-	**	*	*	-	-	ID-R23	ID-R23	**	...	22
R50 TransformValueToBooleanLiteral	*	*	*	-	**	*	*	**	**	ID-R23	ID-R23	**	...	22
R49 TransformValueToCharacterLiteral	*	*	*	-	**	*	*	-	-	ID-R23	ID-R23	**	...	20
R31 TransformateAddressesSingleVariableAccess	*	*	*	-	**	*	*	-	-	-	-	-	...	15
R51 TransformActionElementToNullLiteral	*	*	*	-	**	*	*	-	-	-	-	-	...	11
R52 TransformActionElementToThisExpression	*	*	*	-	**	*	*	-	-	-	-	-	...	10
R46 TransformActionElementToSuperMethodInvocation	*	*	*	-	-	*	*	-	-	-	-	-	...	7
R10 TransformInterfaceUnitToAnnotationTypeDeclaration	*	*	*	-	-	*	-	-	-	-	-	-	...	6
R13 TrasformStorableUnitToFieldDeclaration	*	*	**	*	-	-	-	-	-	-	-	-	...	6

2. Identificar Regras *Folha*: Como foi dito, as regras consideradas *folhas* apresentam dependências de outras regras, mas nenhuma regra depende dela. Essas regras ajudam no propósito de exercitar a maior quantidade de regras de transformação no menor número de casos de teste. Para isto, somamos as ocorrências (qualquer tipo de dependência) nas colunas da matriz e identificamos as regras com resultado igual a zero ("0"). Na Tabela 6.1 percebemos que o R13 (gerar atributos) é uma regra *Folha*.

3. Priorizar das Regras *Folha*: após a identificação das regras *Folhas*, geramos uma nova matriz considerando as regras *Folhas* na vertical e mantendo as regras de transformação na horizontal. Em seguida, contamos as ocorrências nas linhas e reordenamos as regras considerando as maiores no topo. Esses valores indicam numericamente o grau de dependência e ajudam a exercitar um maior número de regras de transformação. Tabela 6.2 mostra parte das regras *folha*, podemos observar o número total de dependências para cada regra. A regra R47 (Criar constante numérica) tem como resultado o valor de '25', isto significa que a utilização da regra R47 pode exercitar até 25 regras de transformação do conjunto de 52.

4. Definir Dados de Teste: após a priorização das regras *Folhas*, optamos por combiná-las com o objetivo de escolher programas Java completos (ver Apêndice D). A Tabela 6.3 apresenta o resumo dos sete casos de teste, as regra *folha* que atinge e a quantidade de regras cobertas por cada folha. Observa-se, o caso de Teste I atinge as regras *folha* R47, R30 e R51 e cada uma delas exercita certo número de regras das quais dependem. Assim, a regra *folha* R47 ajuda na execução (pelo menos uma vez) de 10 regras de transformação, a regra *folha* R30 executa 7 novas regras e a regra *folha* R51 executa mais 2 regras de transformação. Como resultado o Caso de Teste I vai testar 19 regras de transformação que representam 36.51% do total. O conjunto dos sete casos de teste garantem a execução das 52 regras de transformação da ferramenta.

Tabela 6.3 – Casos de Teste

CT [1]	**R47**	**R30**	**R48**	**R50**	**R49**	**R31**	**R51**	**R52**	**R46**	**R10**	**R13**	**C** [2]	**P%** [3]
I	10	7	-	-	-	-	2	-	-	-	-	19	36.54
II	3	-	-	-	-	5	-	-	-	-	-	8	15.38
III	-	2	3	2	-	-	-	-	-	-	-	7	13.46
IV	2	-	-	-	-	-	-	2	1	-	-	5	9.62
V	-	-	-	-	-	-	-	-	-	2	2	4	7.69
VI	-	-	-	-	-	5	-	-	-	-	-	5	9.62
VII	-	3	-	-	1	-	-	-	-	-	-	4	7.69
Total:												52	100

[1] CT: Caso de Teste
[2] C: Cobertura, refere-se a quantidade de regras exercitadas
[3] P: Porcentagem de Cobertura

Tabela 6.4 – Regras folhas em cada caso de Teste

	Casos de Teste - Detalhe															
	I			II		III			IV			V		VI	VII	
Regra	**R47**	**R30**	**R51**	**R47**	**R31**	**R30**	**R48**	**R50**	**R47**	**R52**	**R46**	**R10**	**R13**	**R31**	**R30**	**R49**
R01	*															
R02						*										
R03					*											
R04														*		
R05	*															
R06	-															
R07	-															
R08	-															
R09												*				
R10												*				
R11	*															
R12													*			
R13													*			
R14									*							
R15	*															
R16	*															
R17	*															
R18		*														
R19	*															
R20								*								
R21		*														
R22		*														
R23		*														
R24							*									
R25							*									
R26									*							
R27	*															
R28										*						
R29	*															
R30		*														
R31					*											
R32															*	
R33			*													
R34															*	
R35				*												
R36															*	
R37		*														
R38				*												
R39				*												
R40		*														
R41						*										
R42					*											
R43														*		
R44					*											
R45					*											
R46											*					
R47	*															
R48							*									
R49																*
R50								*								
R51			*													
R52										*						
R53														*		
R54														*		
R55														*		
Total	10	7	2	3	5	2	3	2	2	2	1	2	2	5	3	1

A Tabela 6.4 apresenta maior granularidade na identificação das regras exercitadas por cada *folha* que compõe cada caso de Teste. Observa-se, na horizontal da matriz os casos de teste com as regras *folha* atingidas y na vertical as regras de transformação exercitadas.

No caso de Teste I, que é composto pela combinação das regras *folha* R47, R30 e R51 e pelo programa Java chamado de *StringSwitchDemo* (Ver Apêndice D), observa-se que a regra *folha* R47 executa as regras R01, R05, R11,R15, R16, R17, R19, R27, R29 e R47 totalizando 10 regras de transformação. Além disso, a regra *folha* R30 executa as regras R18, R21, R22, R23, R30, R37 e R40, totalizando 7 regras de transformação. Por fim, a regra *folha* R51 executa mais 2 regras as quais são R33 e R51. Como resultado 19 regras de transformação são validadas com a execução do caso de Teste I.

Note-se, que as regras são contabilizadas tendo em consideração a primeira ocorrência, isto é, muitas regras são executadas mais de uma vez em cada caso de teste. Todos os casos de Teste escolhidos são apresentados no Apêndice D e o calculo da quantidade de regras de transformação executadas por cada regra *folha* foi realizado manualmente.

6.4 Executar Caso de Teste

Após a elaboração dos sete casos de teste, programas Java disponíveis no Apêndice D, e com a identificação das regras de transformação a serem testadas se procede com a execução. A Figura 6.2 apresenta o processo de execução dos Casos de Teste e as atividades envolvidas, que são descritas a seguir.

Primeiro, na atividade *'A'* gera-se o Modelo KDM a partir do programa Java escolhido (Apêndice D). Para isto, utiliza-se o suporte do Framework Eclipse e do projeto ATL *Discover-Advanse*. Como foi dito anteriormente, corrigiu-se algumas regras de transformação no projeto ATL do Modisco, mas que ainda não foram disponibilizadas pelo projeto Eclipse. Sendo assim, se providencia o projeto *Discover KDM-Advanse* que permite gerar modelos KDM refinados, superando os defeitos, até agora detectados, na ferramenta Modisco.

Em seguida, na atividade *'B'* gera-se o Modelo Java a partir do Modelo KDM, gerado na atividade A, com suporte do motor de transformação RUTE-K2J. Nesta transformação se espera que a informação seja preservada, conservando as estruturas de código-fonte contidas no código fonte de origem.

Posteriormente, na atividade *'C'* gera-se código fonte a partir do Modelo Java, saída da ferramenta RUTE-K2J, com suporte do plug-in Acceleo que é uma ferramenta de transformação modelo para texto que apresenta um tipo de abordagem baseado em Templates. Esse tipo de abordagens é a mais utilizada e conhecida pelos desenvolvedores.

Por fim, na atividade *'D'* compara-se o Programa Java origem (ver Apêndice D) com o código fonte gerado pela ferramenta Acceleo com o objetivo de verificar a similaridade dos

modelos, que indicará o grau da preservação da informação durante a transformação. Para isto, se utiliza o apoio da ferramenta livre *Pretty Diff*.

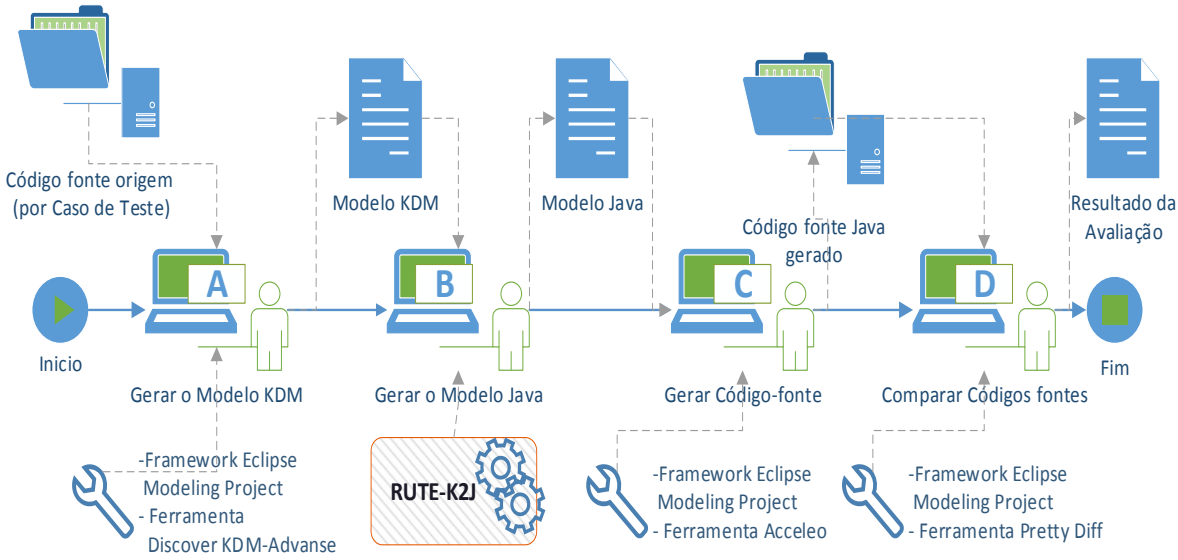

Figura 6.2 – Execução dos Casos de Teste

6.5 Análise do Resultado da Avaliação

O objetivo da avaliação é visar a corretude de todas as regras de transformação que compõem o motor RUTE-K2J, isto é, se as regras cumprem com o objetivo para as que foram criadas, por exemplo gerar estruturas *for*, *if* e atribuição de variáveis. Além disso, verificar se as regras em conjunto transformam corretamente estruturas complexas.

A execução dos casos de teste fornece como resultado o grau de preservação da informação expressado em linhas de código levantando indícios da qualidade da transformação e por consequência a qualidade das regras de transformação que compõem o motor RUTE-K2J.

Na Tabela 6.5 são mostrados os resultados. Observa-se que a primeira coluna indica o número do caso de Teste, a segunda coluna indica o nome do projeto de código fonte, a terceira coluna indica as linhas de código do programa, a quarta coluna indica a quantidade de linhas de código diferentes e a quinta e sexta coluna apresentam uma comparativa entre a linha de código original e a linha de código com diferença. Como resultado, dos sete casos de teste com um total de 217 linhas de código, 201 linhas foram geradas corretamente e 16 linhas apresentam diferenças, isto é, 92% do código foi gerado com sucesso.

As linhas de código gerados com diferenças são analisadas a seguir:

- Para todos os casos de teste, a palavra reservada *Static* não está sendo gerada. Esse dado não é trazido pelo modelo de entrada KDM gerado pelo *Discover-Advanse* nem pelo

Tabela 6.5 – Resultado da Execução dos Casos de Teste

CT[1]	Projeto Java	L [2]	LD [3]	Código fonte origem	Código fonte Acceleo[4]

I	ControlFlowStatements.java	61	4	public *static* int getMonthNumber
				public *static* void main(...
				if (month == null)
				if (returnedMonthNumber == 0)
					public int getMonthNumber
					public void main(...
					if (null == month)
					if (0 == returnedMonthNumber)

II	TestArray.java	20	1	public *static* void main(...	public void main(...
III	GetAge.java	21	2	public *static* void main(... sc = new Scanner(*System.in*)	public void main(... sc = new Scanner();
IV	TestBikes.java MountainBike.java Bicycle.java	11 20 28	1 0 0	public *static* void main(... - -	public void main(... - -
V	AnonymousDemo.java Age.java	13 6	2 0	public *static* void main(... } -	public void main(... }(); -
VI	ArrayListTOArray.java	18	2	public *static* void main(... String array[] = new ...	public void main(... String array = new ...

VII	Test.java	19	4	//Java program to ...
				public *static* void main(...
				b = (byte)(b * 2);
				println("Prefix = "+ ++i);
					-
					public void main(...
					b = (byte)(2 * b);
					println("Prefix = "+++i);

Total		217	16		

[1] CT: Caso de Teste
[2] L: Líneas de Código
[3] LD: Líneas de Código diferentes
[4] Código com diferenças

Discover da ferramenta Modisco. O metamodelo KDM será revisado para identificar o atributo que deveria armazenar essa informação.

- Caso de Teste I. Na estrutura de controle *if-then-else*, os elementos da parte condicional apresentam diferenças na ordem de aparição. Esse é um dos problemas enfrentados que foi descrito na seção 5.3.

- Caso de teste III. Na criação da classe *Scaner* o parâmetro *System.in* está ausente. Esse problema foi detectado durante o desenvolvimento da ferramenta RUTE-K2J. O modelo KDM não está trazendo essa informação, revisando a documentação do metamodelo não foi encontrado exemplos claros de como essa informação deve ser representada no modelo.

- Caso de Teste V. Após a criação de uma classe anônima está sendo gerado um parêntese *"()"* desnecessário. Essa falha será corregida para uma seguinte versão da ferramenta RUTE-K2J.

- Caso de teste VI. Na criação de uma estrutura *Array []* o elemento *"[]"* está ausente. A informação está ausente no modelo de entrada KDM. O metamodelo KDM será revisado para identifica o atributo que deveria armazenar essa informação.

- Caso de teste VII. Os comentários não estão sendo gerados. Será desenvolvida uma regra de transformação com esse proposito para uma próxima versão da ferramenta. Além disso, na expressão *(2 * b)* a ordem dos elementos está incorreta. Esse é o mesmo problema reportado para o *Test Case I*. Também a expressão *"+++i"* não mostra a separação

entre os elementos *"+"* e *"++i"*. Neste ultimo caso, o problema está na ferramenta Acceleo ao gerar código fonte.

Como resultado da avaliação, 92% do código foi gerado com sucesso. Isso mostra que as regras de transformação, apesar das limitações do modelo de entrada KDM, geram um modelo Java que preserva a maior parte das estruturas e dados incorporadas no código-fonte, fornecendo indícios de qualidade nas regras de transformação e do Motor de transformação RUTE-K2J.

6.6 Ameaças à Validade

O pequeno número de exemplos de código fonte coletado. No entanto, quando se procurava pelas amostras de código-fonte, tentou-se obter amostras representativas, ou seja, aquelas que apresentavam estruturas de código fonte comuns e usuais;

Todas as combinações entre regras não foram exercidas. Embora se tenha exercido as regras em muitas combinações em cada caso de teste, muitas foram deixadas de lado. Esta é uma ameaça que se pretende resolver em um trabalho futuro.

Todas as combinações de valores que podem adotar os atributos não foram exercidas. As amostras de código-fonte fornecem valores para os atributos das metaclasses, mas não garantem testar todos os possíveis valores que podem adotar cada um deles. Esta é uma ameaça que pretendemos também resolver em um trabalho futuro.

6.7 Considerações Finais

Neste capítulo foi apresentada a avaliação deste trabalho de pesquisa que é baseada na utilização de Casos de Teste e guiada sob a estratégia de validar a corretude das regras de transformação que compõem RUTE-K2J. Para isto, foram detalhadas as atividades para projetar, implementar e executar cada caso de teste.

Na projeção dos Casos de teste, foi apresentado o processo conformado por 4 atividades que tem como objetivo descrever os passos para seleção dos dados de teste (programas Java) e do subconjunto de regras de transformação a serem testadas. Já na implementação, os passos descritos são concretados com a criação da Matriz de dependências, Matriz de folhas, a seleção dos programas Java e do subconjunto de regras de transformação a serem testadas.

Além disso, foram detalhados os 4 passos para a execução dos casos de teste, que envolve a utilização de ferramentas como: Eclipse, Discover KDM-Advanse, Acceleo, e Pretty Diff. Como resultado da execução, se verifica que RUTE-K2J é capaz de gerar um 92% do código de entrada, o que indica um alto grau de qualidade nas regras de transformação desenvolvidas.

Capítulo 7
CONCLUSÃO

O último capítulo deste trabalho de pesquisa visa estabelecer conclusões, delinear as limitações presentes e sugerir algumas recomendações de trabalhos futuros relacionados ao tema principal.

Nesta dissertação, almejou-se fornecer uma solução para o estágio de engenharia avante dos projetos de modernização ADM. Para isso, foi criado o apoio ferramental chamado RUTE-K2J que permite as transformações do modelo KDM para modelo Java. Assim como, uma abordagem para a criação de motores de transformação do modelo KDM para modelos PSMs, que foi criada a partir da generalização do processo seguido durante o desenvolvimento do motor RUTE-K2J. Como parte da pesquisa, uma avaliação à ferramenta RUTE-K2J foi conduzida com o intuito de avaliar a corretude das transformações.

Do desenvolvimento do trabalho de pesquisa pode-se inferir o seguinte:

- A abordagem proposta permite a construção de motores de transformação do modelo KDM para PSMs. Portanto, disponibilizá-la fornece uma guia para os engenheiros de modernização na criação de motores de transformação e, por consequência, completar o estágio de engenharia avante dos projetos de modernização ADM. A abordagem é composta por 4 fases chamadas: Escolher PSM alvo e Ferramenta de Engenharia Reversa, Gerar Artefatos Iniciais, Desenvolver Regra de Transformação e Validar Regra de Transformação. Além disso, caracterizada por ser iterativa e incremental.

- A ferramenta RUTE-K2J, versão inicial, fornece um instrumento para transformações exógenas e verticais do modelo KDM para o modelo Java. Portanto, disponibilizá-la permite a reutilização da ferramenta em diferentes casos de uso de modernização. Assim como, ser estendia e continuada por interessados permitindo abordar casos de uso de modernização de maior complexidade.

- A construção da ferramenta RUTE-K2J foi caracterizada pelas ações para garantir a preservação da informação. Isto é, lidou-se com o diferente nível de abstração dos modelos e com o modelo de entrada KDM incompleto. As características das dificultadas enfrentadas e as soluções podem ser revisadas no Capítulo 5.

- A qualidade da ferramenta, RUTE-K2J foi avaliada com *Casos de Teste* com o objetivo de verificar a corretude das regras de transformação desenvolvidas que compõem a ferramenta. O planejamento, implementação e execução dos sete casos de teste são resultado de um processo funcional definido para o contexto do trabalho de pesquisa. A avaliação mostrou que 92% do código fonte foi preservado, fornecendo indícios da qualidade das regras de transformação desenvolvidas e por consequência da qualidade do motor de transformação RUTE-K2J. O 8% do código não gerado é causado principalmente pelo modelo KDM de entrada com problemas na preservação da informação ao longo do processo de reengenharia.

7.1 Limitações

A abordagem e o apoio computacional desenvolvidos apresentam as seguintes limitações:

- A abordagem de criação de motores de transformação KDM para PSMs deve ser executada manualmente. Como consequência, uma maior responsabilidade é atribuída ao engenheiro de modernização.
- A abordagem precisa de suportes ferramentais que gerem o modelo KDM e PSM. Em razão que na abordagem, a criação do mapeamento é baseado na análise das instâncias do metamodelo KDM e do metamodelo PSM escolhido.
- O motor de transformação RUTE-K2J, versão inicial, pode transformar uma quantidade limitada de estruturas. Isto é, o motor Rute-K2J foi desenvolvido segundo os *Requisitos de Transformação* e pelos trechos de código fornecidos como entrada pelo engenheiro de modernização. Por tanto, o motor só reconhece essas estruturas como entrada e somente para elas apresenta uma regra de transformação ATL.
- O motor RUTE-K2J, versão inicial, é um projeto ATL desenvolvido para ser utilizado no ambiente de desenvolvimento Eclipse.

7.2 Trabalhos futuros

Como sugestão de trabalhos futuros, pode-se mencionar o seguente:

- Automatizar atividades na abordagem de criação de motores de transformação KDM para PSM. Isto é, algumas atividades ou parte de elas podem ser automatizadas para fornecer maior apoio ao engenheiro de modernização. Por exemplo: A criação de um mapeamento inicial, gerar um esboço da regra de transformação, comparação dos Modelos XMI;

- Integrar a validação das regras em conjunto como una fase dentro da abordagem de criação de motores de transformação KDM para PSM. Assim como demostrar o processo iterativo da abordagem com um caso de estudo.

- Realizar uma revisão sistemática da literatura de abordagens para a criação de motores de transformação que considerem transformações exógenas e verticais, com o objetivo de evoluir a abordagem que foi proposta neste trabalho de pesquisa;

- Evoluir a ferramenta RUTE-K2J, como dito nas limitações, a ferramenta ainda não fornece suporte para transformar todos os tipos de estruturas e as variações que possam ter. Para isto, RUTE-K2J deve ser continuada com o estabelecimento de novos *Requisitos de Transformação* e trechos de código;

- Integrar a ferramenta RUTE-K2J com uma ferramenta M2T, com o intuito de potenciar a solução de Engenharia Avante fornecida ao engenheiro de modernização. Uma sugestão de trabalho relacionado seria a integração e automação de duas transformações de KDM-para-PSM e PSM-para-Código;

- Avaliar o motor de transformação RUTE com outras técnicas de teste, além da apresentada neste projeto de pesquisa. Uma opção é um tipo de avaliação mais funcional com a execução do código fonte gerado pela ferramenta Acceleo.

.

REFERÊNCIAS

BARBIER, F.; DELTOMBE, G.; PARISY, O.; YOUBI, K. Model driven reverse engineering: Increasing legacy technology independence. In: *Second India Workshop on Reverse Engineering*. [S.l.: s.n.], 2011. v. 125, p. 126–139.

BENNETT, K. Legacy systems: Coping with success. *IEEE software*, v. 12, n. 1, 1995.

BRUNELIERE, H.; CABOT, J.; DUPÉ, G.; MADIOT, F. Modisco: a model driven reverse engineering framework. *Information and Software Technology*, Elsevier, v. 56, n. 8, p. 1012–1032, 2014.

BRUNELIERE, H.; CABOT, J.; JOUAULT, F.; MADIOT, F. Modisco: a generic and extensible framework for model driven reverse engineering. In: ACM. *Proceedings of the IEEE/ACM international conference on Automated software engineering*. [S.l.], 2010. p. 173–174.

CANOVAS, j.; MOLINA, J. An architecture-driven modernization tool for calculating metrics. *IEEE software*, v. 27, n. 4, 2010.

CHIKOFSKY, E.; CROSS, J. Reverse engineering and design recovery: A taxonomy. *IEEE software*, v. 7, n. 1, 1990.

CZARNECKI, K.; HELSEN, S. Feature-based survey of model transformation approaches. *IBM Systems Journal*, IBM, v. 45, n. 3, 2006.

ECLIPSE. *Eclipse ATL*. 2005. Disponível em: <https://projects.eclipse.org/projects/modeling.mmt.atl>. Acesso em: 2018-10-28.

ECLIPSE. *Acceleo*. 2006. Disponível em: <http://www.eclipse.org/acceleo/>. Acesso em: 2018-03-28.

ECLIPSE. *Modisco*. 2006. Disponível em: <https://www.eclipse.org/MoDisco/>. Acesso em: 2018-04-28.

GOURSHETTIWAR, P.; BHANDARIANISH, A. P.; SHIRBHATE, D. D. Modernised architecture. *International Journal For Engineering Applications And Technology*, 2014.

KAHANI, N.; CORDY, J. R. Comparison and evaluation of model transformation tools. *Technical Report 2015-627*, 2015.

KALLIAMVAKOU, E.; GOUSIOS, G.; BLINCOE, K.; SINGER, L.; GERMAN, D. M.; DAMIAN, D. The promises and perils of mining github. In: ACM. *Proceedings of the 11th working conference on mining software repositories*. [S.l.], 2014. p. 92–101.

KLEPPE, A.; WARMER, J.; BAST, W. *MDA Explained: The Model Driven Architecture: Practice and Promise*. [S.l.]: Addison-Wesley Professional, 2003.

LEHMAN, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, v. 68, n. 9, 1980.

MARTINEZ, L.; PEREIRA, C.; FAVRE, L. Recovering sequence diagrams from object-oriented code: An adm approach. In: IEEE. *Evaluation of Novel Approaches to Software Engineering (ENASE), 2014 International Conference on*. [S.l.], 2014. p. 1–8.

MENS, T.; GORP, P. V. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 152, p. 125–142, 2006.

MORATALLA, J.; CASTRO, V. de; SANZ, M. L.; MARCOS, E. A gap-analysis-based framework for evolution and modernization: Modernization of domain management at red. es. In: IEEE. *Srii global conference (srii), 2012 annual*. [S.l.], 2012. p. 343–352.

OMG. *MDA- The Architecture of Choise for a Changing World*. 2001. Disponível em: <https://www.omg.org/mda/>. Acesso em: 2018-09-28.

OMG. *Object Constraint Language*. 2006. Disponível em: <https://www.omg.org/spec/OCL/2.4/>. Acesso em: 2018-07-28.

OMG. *Architecture-Driven Modernization*. 2009. <http://www.omgwiki.org/admtf/doku.php?id=start>. Acesso em: 2018-06-28.

OMG. *About the Knowledge Discovery Metamodel Specification*. 2016. Disponível em: <https://www.omg.org/spec/KDM/>. Acesso em: 2018-08-28.

PÉREZ-CASTILLO, R.; CRUZ-LEMUS, J. A.; GUZMÁN, I. G.-R. de; PIATTINI, M. A family of case studies on business process mining using marble. *Journal of Systems and Software*, Elsevier, v. 85, n. 6, p. 1370–1385, 2012.

PÉREZ-CASTILLO, R.; GUZMÁN, I. D.; AVILA-GARCIA, O.; PIATTINI, M. On the use of adm to contextualize data on legacy source code for software modernization. In: IEEE. *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. [S.l.], 2009. p. 128–132.

PÉREZ-CASTILLO, R.; GUZMÁN, I. D.; PIATTINI, M. Implementing business process recovery patterns through qvt transformations. In: SPRINGER. *International Conference on Theory and Practice of Model Transformations*. [S.l.], 2010. p. 168–183.

PÉREZ-CASTILLO, R.; GUZMÁN, I. G. R. de; GÓMEZ-CORNEJO, R.; FERNANDEZ-ROPERO, M.; PIATTINI, M. Andriu. a technique for migrating graphical user interfaces to android (s). In: *SEKE*. [S.l.: s.n.], 2013. p. 516–519.

PÉREZ-CASTILLO, R.; GUZMÁN, I. G. R. de; PIATTINI, M. Architecture-driven modernization. *Modern Software Engineering Concepts and Practices: Advanced Approaches: Advanced Approaches*, IGI Global, v. 75, 2010.

PÉREZ-CASTILLO, R.; GUZMÁN, I. G.-R. de; PIATTINI, M. Business process archeology using marble. *Information and Software Technology*, Elsevier, v. 53, n. 10, p. 1023–1044, 2011.

PÉREZ-CASTILLO, R.; GUZMÁN, I. G.-R. de; PIATTINI, M. Mimos, system model-driven migration project. In: IEEE. *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. [S.l.], 2013. p. 445–448.

PÉREZ-CASTILLO, R.; GUZMÁN, I. García-Rodríguez de; PIATTINI, M.; PLACES, A. S. A case study on business process recovery using an e-government system. *Software: Practice and Experience*, Wiley Online Library, v. 42, n. 2, p. 159–189, 2012.

RODRÍGUEZ-ECHEVERRÍA, R.; CONEJERO, J. M.; CLEMENTE, P. J.; PRECIADO, J. C.; SÁNCHEZ-FIGUEROA, F. Modernization of legacy web applications into rich internet applications. In: SPRINGER. *International Conference on Web Engineering*. [S.l.], 2011. p. 236–250.

TRIAS, F.; CASTRO, V. D.; LÓPEZ-SANZ, M.; MARCOS, E. A toolkit for adm-based migration: Moving from php code to kdm model in the context of cms-based web applications. 2014.

ULRICH, W. M.; NEWCOMB, P. *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. [S.l.]: Morgan Kaufmann, 2010.

VASILECAS, O.; NORMANTAS, K. Deriving business rules from the models of existing information systems. In: ACM. *Proceedings of the 12th International Conference on Computer Systems and Technologies*. [S.l.], 2011. p. 95–100.

VISSER, E. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 57, p. 109–143, 2001.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *Experimentation in Software Engineering*. [S.l.]: Springer, 2012. 236 p.

WULF, C.; FREY, S.; HASSELBRING, W. A three-phase approach to efficiently transform c# into kdm. Department of Computer Science, Kiel University, Germany, 2012.

SIGLAS

ADM	Architecture-Driven Modernization
ADMPR	ADM Pattern Recognition
ADMRS	ADM Refactoring Specification
ADMTF	Architecture-Driven Modernization Task Force
ADMVS	ADM visualization Specification
AST	Abstract Syntax Tree
ASTM	Abstract Syntax Tree Metamodel
ATL	Atlas Transformation Language
BPM	Business Process Model
CIM	Computation Independent Model
CMS	Content Management Systems
CSV	Comma-Separated Values
KDM	Knowledge Discovery Metamodel
LHS	Left Hand Side
M2M	Model to Model
M2T	Model to Text
MDRE	Model Driven Reverse Engineering
OMG	Object Management Group
PIM	Platform Independent Model
PL/SQL	Procedural Language
PSM	Platform Specific Model
QVTo	Query/View/Transformation Operational
RFP	Request for proposal
RHS	Right Hand Side
RIA	Rich Internet Applications
ROI	Return on Investment
SDK	Software Development Kit
SMM	Structured Metrics MetaModel
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XSLT	Xtensible Style-sheet Language Transformation

Apêndice A

MAPEAMENTO COMPLETO KDM-PSM

Tabela A.1 – Mapeamento Completo KDM2JAVA

Modelo KDM			**Modelo JAVA**
Pacote	**Metaclasse**	**Filtro**	**Metaclasse**
Code	CodeModel (main) CodeModel (External)	- - - -	Model
Code	AbstractCodeElement	-	BodyDeclaration
Code	TemplateUnit	-	TypeDeclaration
Code	Datatype	-	AbstractTypeDeclaration
Code	LanguageUnit	-	PrimitiveType
Code	Package	-	Package
Code	ClassUnit	name <> 'Anonymous type' AND ContenedorSuperior <> 'TemplateUnit' name = 'Anonymous type'	ClassDeclaration AnonymousClassDeclaration
Code	InterfaceUnit	ContenedorSuperior() <> TemplateUnit elemento 'annotation' <> Empty()	InterfaceDeclaration AnnotationTypeDeclaration
Code	TemplateUnit	getRealTypeClassUnit() = True getRealTypeInterfaceUnit() =True	ClassDeclaration InterfaceDeclaration
Code	MethodUnit	kind = constructor kind = method	ConstructorDeclaration MethodDeclaration
Code	StorableUnit	kind <> local kind = local	FieldDeclaration VariableDeclarationFragment
Code	TemplateParameter	-	TypeParameter
Code	TemplateType	-	ParameterizedType
Code	ArrayType	-	ArrayType
Code	ParameterUnit	kind = throws kind = return kind = unknown	TypeAccess SingleVariableDeclaration
Code	Implements		TypeAccess
Code	Extends		TypeAccess
Code	Imports		ImportDeclaration
Code	ArrayType		ArrayType
Code	Value	name='number literal' name='string literal' name='character literal' name='boolean literal'	NumberLiteral StringLiteral CharacterLiteral BooleanLiteral

Continua na próxima página

Tabela A.1- Continuação da página anterior

Modelo KDM			**Modelo JAVA**
Pacote	**Metaclasse**	**Filtro**	**Metaclasse**
Source	InventoryModel	-	CompilationUnit
Action	BlockUnit	-	Block
Action	ActionElement	kind =’infix expression’	InfixExpression
		kind=’postfix expression’	PostfixExpression
		kind=’while’	WhileStatement
		kind=’break’	BreakStatement
		kind=’case’	SwitchCase
		kind=’switch’	SwitchStatement
		kind=’cast’	CastExpression
		kind=’expression statement’	ExpressionStatement
		kind=’return’	ReturnStatement
		kind =’variable declaration’ and ContenedorSuperior() <> ’For’"	VariableDeclarationStatement
		kind =’variable declaration’ and ContenedorSuperior() = ’For’"	VariableDeclarationExpression
		kind =’assignment’	Assignment
		kind=’field access’	FieldAccess
		kind = ’parenthesized’	ParenthesizedExpression
		kind= ’prefix expression’	PrefixExpression
		kind=’this’	ThisExpression
		kind=’method invocation’	MethodInvocation
		kind=’for’	ForStatement
		kind=’if’	IfStatement
		kind=’class instance creation’	ClassInstanceCreation
		kind=’array access’	ArrayAccess
		kind=’array creation’	ArrayCreation
		kind =’array initializer’	ArrayInitializer
		kind= ’array length access’	ArrayLengthAccess
		kind= ’null’	NullLiteral
		kind =’super constructor invocation’	SuperConstructorInvocation
		kind= ’super method invocation’	SuperMethodInvocation
Action	CatchUnit	Quando herda de ’ExceptionUnit’	CatchClause
Action	TryUnit	Quando herda de ’ExceptionUnit’	TryStatement
Action	Writes		SingleVariableAccess
Action	Reads		SingleVariableAccess
Action	Addresses		SingleVariableAccess

Apêndice B

INVENTÁRIO DE REGRAS DE TRANSFORMAÇÃO

Tabela B.1 – Regras de Transformação

Nº	Nome da Regra	Objetivo
R01	TransformCodeModelToJavaModel	Regra principal para estruturar o Modelo Java a partir do Modelo KDM
R02	TransformImportsToImportDeclaration	Transformar as metaclasses 'Imports' do Modelo KDM para metaclasses 'ImportDeclaration' do Modelo Java
R03	TransformArrayTypeToArrayType	Transformar as metaclasses 'ArrayType' do Modelo KDM para metaclasses 'ArrayType' do Modelo Java
R04	TransformTemplateTypeToParameterizedType	Transformar as metaclasses 'TemplateType' do Modelo KDM para metaclasses 'ParameterizedType' do Modelo Java
R05	TransformPackageToJavaPackage	Transformar a metaclasse 'Package' do metamodelo KDM para metaclasses 'Packages' do metamodelo Java
R06	TransformAbstractCodeElementToBodyDeclaration	Regra Abstrata
R07	TansforDatatypeToAbstractTypeDeclaration	Regra Abstrata
R08	TransformTemplateUnitToTypeDeclaration	Regra Abstrata
R09	TransformClassUnitToAnonymousClassDeclaration	Transformar as metaclasses 'ClassUnit' do metamodelo KDM para metaclasses 'AnnotationTypeDeclaration' no Modelo Java
R10	TransformInterfaceUnitToAnnotationTypeDeclaration	Transformar as metaclasses 'InterfaceUnit' do Modelo KDM para metaclasses 'AnnotationTypeDeclaration' do Modelo Java
R11	TransformClassUnitToClassDeclaration	Transformar as metaclasses 'ClassUnit' do metamodelo KDM para metaclasses 'ClassDeclaration' do metamodelo Java
R12	TransformInterfaceUnitToInterfaceDeclaration	Transformar as metaclasses 'InterfaceDeclaration' do Modelo KDM para metaclasses 'InterfaceUnit' do Modelo Java
R13	TrasformStorableUnitToFieldDeclaration	Transformar as metaclassses 'StorableUnit' do Modelo KDM para metaclasses 'FieldDeclaration' do Modelo Java
R14	TransformMethodUnitToConstructorDeclaration	Transformar as metaclasses 'MethodUnit' do tipo 'constructor' do metamodelo KDM para metaclasses 'ConstructorDeclaration' do metamodelo Java
R15	TransformMethodUnitToMethodDeclaration	Transformar as metaclasses 'MethodUnit' do tipo 'method' do metamodelo KDM para metaclasses 'ConstructorDeclaration' do metamodelo Java

Continua na próxima página

Tabela B.1 Continuação da página anterior

Nº	Nome da Regra	Objetivo
R16	TransformBlockToBlockUnit	Transformar as metaclasses 'BlockUnit' do Modelo KDM para metaclasses 'Block' do Modelo Java
R17	TransformActionElementToExpressionStatement	Transformar as metaclasses 'ActionElement' do tipo 'assignment' do Modelo KDM para metaclasses 'Assignment' do Modelo Java
R18	TransformActionElementToReturnStatement	Transformar as metaclasses 'ActionElement' do tipo 'return' do Modelo KDM para metaclasses 'ReturnStatement' do Modelo Java
R19	TransformActionToVariableDeclarationStatement	Transformar as metaclasses 'ActionElement' do tipo 'variable declaration' do Modelo KDM para metaclasses 'VariableDeclarationStatement' do Modelo Java
R20	TransformActionElementToWhileStatement	Transformar as metaclasses 'ActionElement' do tipo 'while' do Modelo KDM para metaclasses 'WhileStatement' do Modelo Java
R21	TransformActionElementToSwitchStatement	Transformar as metaclasses 'ActionElement' do tipo 'switch' do Modelo KDM para metaclasses 'SwitchStatement' do Modelo Java
R22	TransformActionElementToBreakStatement	Transformar as metaclasses 'ActionElement' do tipo 'break' do Modelo KDM para metaclasses 'BreakStatement' do Modelo Java
R23	TransformActionElementToSwitchCase	Transformar as metaclasses 'ActionElement' do tipo 'case' do Modelo KDM para metaclasses 'SwitchCase' do Modelo Java
R24	TransformTryUnitToTryStatement	Transformar as metaclasses 'TryUnit' do Modelo KDM para metaclasses 'TryStatement' do Modelo Java
R25	TransformCatchUnitToCatchClause	Transformar as metaclasses 'CatchUnit' do Modelo KDM para metaclasses 'CatchClause' do Modelo Java
R26	TransformActionElementToSuperConstructorInvocation	Transformar as metaclasses 'ActionElement' do tipo 'super constructor invocation' do Modelo KDM para metaclasses 'SuperConstructorInvocation' do Modelo Java
R27	TransformActionElementToAssignment	Transformar as metaclasses 'ActionElement' do tipo 'assignment' do Modelo KDM para metaclasses 'Assignment' do Modelo Java
R28	TransformActionElementToFieldAccess	Transformar as metaclasses 'ActionElement' do tipo 'field access' do Modelo KDM para metaclasses 'FieldAccess' do Modelo Java
R29	TransformWritesToSingleVariableAccess	Transformar as metaclasses 'Writes' do Modelo KDM para metaclasses 'SingleVariableAccess' do Modelo Java
R30	TransformReadsToSingleVariableAccess	Transformar as metaclasses 'Reads' do Modelo KDM para metaclasses 'SingleVariableAccess' do Modelo Java
R31	TransformateAddressesSingleVariableAccess	Transformar as metaclasses 'Addresses' do Modelo KDM para metaclasses 'SingleVariableAccess' do Modelo Java
R32	TransformActionElementToCastExpression	Transformar as metaclasses 'ActionElement' do tipo 'cast' do Modelo KDM para metaclasses 'CastExpression' do Modelo Java
R33	TransformActionElementToInfixExpression	Transformar as metaclasses 'ActionElement' do tipo 'infix expression' do Modelo KDM para metaclasses 'InfixExpression' do Modelo Java

Continua na próxima página

Tabela B.1 Continuação da página anterior

Nº	Nome da Regra	Objetivo
R34	TransformActionElementToParenthesizedExpression	Transformar as metaclasses 'ActionElement' do tipo 'parenthesized' do Modelo KDM para metaclasses 'ParenthesizedExpression' do Modelo Java
R35	TransformActionElementToPostfixExpression	Transformar as metaclasses 'ActionElement' do tipo 'postfix expression' do Modelo KDM para metaclasses 'PostfixExpression' do Modelo Java
R36	TransformActionElementToPrefixExpression	Transformar as metaclasses 'ActionElement' do tipo 'prefix expression' do Modelo KDM para metaclasses 'PrefixExpression' do Modelo Java
R37	TransformActionElementToMethodInvocation	Transformar as metaclasses 'ActionElement' do tipo 'method invocation' do Modelo KDM para metaclasses 'MethodInvocation' do Modelo Java
R38	TransformActionElementToForStatement	Transformar as metaclasses 'ActionElement' do tipo 'for' do Modelo KDM para metaclasses 'ForStatement' do Modelo Java
R39	TransformActionElementToVariableDeclaration Expression	Transformar as metaclasses 'ActionElement' do tipo 'variable declaration' do Modelo KDM para metaclasses 'VariableDeclarationExpression' do Modelo Java
R40	TransformActionElementToIfStatement	Transformar as metaclasses 'ActionElement' do tipo 'if' do Modelo KDM para metaclasses 'IfStatement' do Modelo Java
R41	TransformActionElementToClassInstanceCreation	Transformar as metaclasses 'ActionElement' do tipo 'Class instance creation' do Modelo KDM para metaclasses 'ClassInstanceCreation' do Modelo Java
R42	TransformActionElementToArrayAccess	Transformar as metaclasses 'ActionElement' do tipo 'array access' do Modelo KDM para metaclasses 'ArrayAccess' do Modelo Java
R43	TransformActionElementToArrayCreation	Transformar as metaclasses 'ActionElement' do tipo 'array creation' do Modelo KDM para metaclasses 'ArrayCreation' do Modelo Java
R44	TransformActionElementToArrayInitializer	Transformar as metaclasses 'ActionElement' do tipo 'array initializer' do Modelo KDM para metaclasses 'ArrayInitializer' do Modelo Java
R45	TransformActionElementToArrayLengthAccess	Transformar as metaclasses 'ActionElement' do tipo 'array length access' do Modelo KDM para metaclasses 'ArrayLengthAccess' do Modelo Java
R46	TransformActionElementToSuperMethodInvocation	Transformar as metaclasses 'ActionElement' do tipo 'super method invocation' do Modelo KDM para metaclasses 'SuperMethodInvocation' do Modelo Java
R47	TransformValueToNumberLiteral	Transformar as metaclasses 'Value' do tipo 'number literal' do Modelo KDM para metaclasses 'NumberLiteral' do Modelo Java
R48	TransformValueToStringLiteral	Transformar as metaclasses 'Value' do tipo 'string literal' do Modelo KDM para metaclasses 'StringLiteral' do Modelo Java
R49	TransformValueToCharacterLiteral	Transformar as metaclasses 'Value' do tipo 'character literal' do Modelo KDM para metaclasses 'CharacterLiteral' do Modelo Java
R50	TransformValueToBooleanLiteral	Transformar as metaclasses 'Value' do tipo 'boolean literal' do Modelo KDM para metaclasses 'BooleanLiteral' do Modelo Java

Continua na próxima página

Tabela B.1 Continuação da página anterior

Nº	Nome da Regra	Objetivo
R51	TransformActionElementToNullLiteral	Transformar as metaclasses 'ActionElement' do tipo 'Null' do Modelo KDM para metaclasses 'NullLiteral' do Modelo Java
R52	TransformActionElementToThisExpression	Transformar as metaclasses 'ActionElement' do tipo 'this' do Modelo KDM para metaclasses 'ThisExpression' do Modelo Java
R53	TransformTemplateUnitToClassDeclaration	Transformar as metaclasses 'TemplateUnit' do Modelo KDM para metaclasses 'ClassDeclaration' do Modelo Java
R54	TransformTemplateUnitToInterfaceDeclaration	Transformar as metaclasses 'TemplateUnit' do Modelo KDM para metaclasses 'InterfaceDeclaration' do Modelo Java
R55	TransformTemplateParameterToTypeParameter	Transformar as metaclasses 'TemplateParameter' do Modelo KDM para metaclasses 'TypeParameter' do Modelo Java

Apêndice C

INVENTÁRIO DE HELPERS

Tabela C.1 – Inventario de Helpers

Nº	Nome do Helper	Objetivo
1	getOrphanTypes	Obter uma sequência de elementos 'datatypes' (tipo de dados primitivos) no modelo KDM.
2	getCompilationUnit	Obter o inventario dos artefatos físicos do sistema do modelo KDM
3	getExternalElements	Obter os elementos do Modelo External no modelo KDM
4	getExternalOrphanTypes	Obter uma sequência de elementos 'datatypes' (tipo de dados primitivos) no modelo externo KDM.
5	getParametersMethod	Obter os parâmetros de um Método no modelo KDM
6	getParametersThrows	Obter uma sequência de parâmetros do tipo Throw no modelo KDM
7	getReturns	Obter os retornos de um Método no modelo KDM
8	getOriginalCompilationUnitJavaModel	Retornar a referência da Classe ,no modelo Java, do elemento enviado por parâmetro
9	getOriginalCompilationUnit_auxiliar	Helper que auxilia ao Helper getOriginalCompilationUnitJavaModelKDMmodel
10	getOriginalCompilationUnit JavaModelKDMmodel	Retornar a referência da Classe ao qual pertence o elemento enviado por parâmetro
11	checkElementoExternal	Verificar se o elemento pertence ao Modelo External no modelo KDM
12	getFragment	Obter a variavel (fragment) dos elementos 'Field Declaration', variaveis locais e inicializador da estrutura 'ForStatement'
13	getParametro	Obter as variáveis nos parâmetros dos métodos no modelo Javao
14	getReads	Retornar o elemento 'actionRelation'(KDM) do tipo 'Reads' no modelo KDM
15	getWrites	Retornar o elemento 'actionRelation'(KDM) do tipo 'Writes' no modelo KDM
16	getAddresses	Retorna o elemento 'actionRelation'(KDM) do tipo 'Addresses' no modelo KDM
17	getUsesType	Retorna o elemento 'actionRelation'(KDM) do tipo 'UsesType' no modelo Java
18	getOrdemElementos	Obter a ordem dos elementos no corpo de um método no modelo KDM

Continua na próxima página

Tabela C.1- Continuação da página anterior

Nº	**Nome da Regra**	**Objetivo**
20	getImportsporClasses	Obter os elementos 'imports' de uma Classe no modelo Java
21	getImportsporAnonymusClasses	Obter os elementos 'imports' de uma Classe Anônima no modelo Java
22	checkAnonymus	Helper chamado por o helper getImportsporAnonymus-Classes
23	getPackageCompilationUnit	Retornar a referência do pacote da classe enviada como parâmetro no modelo Java.
24	getTypeCompilationUnit	Obter o tipo de elemento (classe ou Interface) do elementos enviado como parâmetro no modelo Java
25	getRealTypeInterfaceUnit	Identificar se o verdadeiro tipo da metaclasse 'TemplateUnit' é 'interfaceUnit'
26	getRealTypeClassUnit	Identificar se o verdadeiro tipo da metaclasse 'TemplateUnit' é 'ClassUnit'
27	getType	Converter o tipo de dado enviado por parâmetro para o tipo TypeAccess do modelo Java
28	getTypeAccess	Obter a referencia do elemento enviado por parâmetro no modelo Java

Apêndice D

CASOS DE TESTE

D.1 Caso de Teste I

```
1  package dc.ufscar;
2
3  public class StringSwitchDemo {
4
5      public static int getMonthNumber(String month) {
6
7
8          int monthNumber = 0;
9
10         if (month == null) {
11
12             return monthNumber;
13         }
14         switch (month.toLowerCase()) {
15
16             case "january":
17                 monthNumber = 1;
18                 break;
19
20             case "february":
21                 monthNumber = 2;
22                 break;
23             case "march":
24                 monthNumber = 3;
25                 break;
26             case "april":
27                 monthNumber = 4;
28                 break;
29             case "may":
30                 monthNumber = 5;
31                 break;
32             case "june":
33                 monthNumber = 6;
34                 break;
35             case "july":
36                 monthNumber = 7;
37                 break;
38             case "august":
39                 monthNumber = 8;
40                 break;
```

Código D.1 – Código fonte do Caso de Teste I - Parte I

```
1               case "september":
2                   monthNumber = 9;
3                   break;
4               case "october":
5                   monthNumber = 10;
6                   break;
7               case "november":
8                   monthNumber = 11;
9                   break;
10              case "december":
11                  monthNumber = 12;
12                  break;
13              default:
14                  monthNumber = 0;
15                  break;
16          }
17
18          return monthNumber;
19      }
20
21      public static void main(String[] args) {
22
23          String month = "August";
24
25          int returnedMonthNumber =
26              StringSwitchDemo.getMonthNumber(month);
27
28          if (returnedMonthNumber == 0) {
29
30              System.out.println("Invalid month");
31          } else {
32              System.out.println(returnedMonthNumber);
33          }
34  }
35
36  }
```

Código D.2 – Código fonte do Caso de Teste I - Parte II

D.2 Caso de Teste II

```
1 package dc.ufscar;
2
3 public class TestArray {
4
5    public static void main(String[] args) {
6
7       double[] myList = {1.9, 2.9, 3.4, 3.5};
8
9       for (int i = 0; i < myList.length; i++) {
10
11
12
13 "System.out.println(myList[i] + "" "");
14 }"
15
16
17       double total = 0;
18
19       for (int i = 0; i < myList.length; i++) {
20          total += myList[i];
21
22       }
23
24       System.out.println("Total is " + total);
25
26       double max = myList[0];
27
28       for (int i = 1; i < myList.length; i++) {
29
30          if (myList[i] > max) max = myList[i];
31       }
32       System.out.println("Max is " + max);
33    }
34 }
```

Código D.3 – Código fonte do Caso de Teste II

D.3 Caso de Teste III

```
1 package dc.ufscar;
2
3 import java.util.Scanner;
4 import java.util.InputMismatchException;
5
6 class GetAge {
7     private static Scanner sc;
8
9 public static void main(String[] args) {
10
11         sc = new Scanner(System.in);
12         System.out.println("Enter your age: ");
13
14
15         while (true) {
16
17             try {
18
19                 int age = sc.nextInt();
20
21                 System.out.println("Your age is: " + age);
22
23                 break;
24             }
25
26             catch (InputMismatchException e) {
27                 System.out.println("You didn't input a valid age.");
28                 sc.next();
29             }
30         }
31     }
32 }
```

Código D.4 – Código fonte do Caso de Teste III

D.4 Caso de Teste IV

```
1  package dc.ufscar;
2
3  public class Bicycle {
4
5      // the Bicycle class has three fields
6      public int cadence;
7      public int gear;
8      public int speed;
9
10     // the Bicycle class has one constructor
11     public Bicycle() {
12         gear = 0;
13         cadence = 0;
14         speed = 0;
15     }
16
17     // the Bicycle class has four methods
18     public void setCadence(int newValue) {
19         cadence = newValue;
20     }
21
22     public void setGear(int newValue) {
23         gear = newValue;
24     }
25
26     public void applyBrake(int decrement) {
27         speed -= decrement;
28     }
29
30     public void speedUp(int increment) {
31         speed += increment;
32     }
33
34     public void printDescription(){
35         System.out.println("Gear " + this.gear);
36         System.out.println("Cadence of " + this.cadence);
37         System.out.println("Speed of " + this.speed);
38     }
39
40 }
```

Código D.5 – Código fonte do Caso de Teste IV - Parte I

```
1 package dc.ufscar;
2
3 public class MountainBike extends Bicycle {
4     private String suspension;
5
6 "    public MountainBike( int startCadence,int startGear,
7 String suspensionType){"
8         super();
9         cadence = startCadence;
10        gear = startGear;
11        this.setSuspension(suspensionType);
12    }
13
14    public String getSuspension(){
15      return this.suspension;
16    }
17
18    public void setSuspension(String suspensionType) {
19        this.suspension = suspensionType;
20    }
21
22    public void printDescription() {
23        super.printDescription();
24
25        System.out.println("The suspension" + getSuspension());
26    }
27 }
```

Código D.6 – Código fonte do Caso de Teste IV - Parte II

```
1 package dc.ufscar;
2
3 public class TestBikes {
4
5   public static void main(String[] args){
6     Bicycle bike01;
7     Bicycle bike02;
8
9     bike01 = new Bicycle();
10    bike02 = new MountainBike(20, 10, "Dual");
11
12
13    bike01.printDescription();
14    bike02.printDescription();
15
16  }
17 }
```

Código D.7 – Código fonte do Caso de Teste IV - Parte III

D.5 Caso de Teste V

```
1 interface Age
2 {
3     int x = 21;
4     void getAge();
5 }
6 class AnonymousDemo
7 {
8     public static void main(String[] args) {
9
10        // Myclass is hidden inner class of Age interface
11        // whose name is not written but an object to it
12        // is created.
13        Age oj1 = new Age() {
14            @Override
15            public void getAge() {
16                // printing  age
17                System.out.print("Age is "+x);
18            }
19        };
20        oj1.getAge();
21    }
22 }
```

Código D.8 – Código fonte do Caso de Teste V

D.6 Caso de Teste VI

```
1 package dc.ufscar;
2 import java.util.*;
3
4 public class ArrayListTOArray {
5
6 public static void main(String[] args) {
7
8 /*ArrayList declaration and initialization*/
9 ArrayList<String> arrlist= new ArrayList<String>();
10 arrlist.add("String1");
11 arrlist.add("String2");
12 arrlist.add("String3");
13 arrlist.add("String4");
14
15 /*ArrayList to Array Conversion */
16 String array[] = new String[arrlist.size()];
17 for(int j =0;j<arrlist.size();j++){
18   array[j] = arrlist.get(j);
19 }
20 for(int j =0;j<array.length;j++){
21 System.out.println(array[j]);
22 }
23 }
24
25 }
```

Código D.9 – Código fonte do Caso de Teste VI

D.7 Caso de Teste VII

```
1 package dc.ufscar;
2 //Java program to illustrate Type promotion in Expressions
3 class Test
4 {
5   public static void main(String args[])
6   {
7       byte b = 42;
8       char c = 'a';
9       short s = 1024;
10      int i = 50000;
11      float f = 5.67f;
12      double d = .1234;
13
14      // The Expression
15      double result = (f * b) + (i / c) - (d * s);
16
17      //Result after all the promotions are done
18      System.out.println("result = " + result);
19
20      b = (byte)(b * 2);
21      System.out.println(b);
22
23
24      System.out.println("Prefix = " + ++i);
25  }
26 }
```

Código D.10 – Código fonte do Caso de Teste VII