Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Automação de Leis de Refatoração Arquitetural

José Dihego da Silva Oliveira

TRABALHO DE GRADUAÇÃO

Recife, 02 de dezembro de 2008

Universidade Federal de Pernambuco

Centro de Informática

José Dihego da Silva Oliveira

Automação de Leis de Refatoração Arquitetural

Monografia apresentada ao Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Orientador: Augusto Cezar Alves Sampaio

Co-orientador: Rodrigo Teixeira Ramos

Recife, 02 de dezembro de 2008

Aos meus amados pais Genário e Lúcia e fieis irmãos Luana e Igo

Agradecimentos

"*Não existe verdadeira inteligência sem bondade*"
Ludwig van Beethoven

"*Lá no meu pé de serra /Deixei ficar meu coração*
Ai, que saudades tenho /Eu vou voltar pro meu sertão"
Luiz Gonzaga / Humberto Teixeira

"*Em vez de perfume e do luxo da praça,Tem chêro sem graça de amargo suó,Suó de cabôclo que vem do roçado,Com fome, cansado e queimado do só.*"
Inspiração nordestina,Patativa do Assaré

Deus. Agradeço a Deus em primeiro lugar pela dádiva maravilhosa de ter cumprido mais um ciclo na minha vida. Deus, fonte de coragem e fé, que me levaram até este ponto na minha existência.

Agradeço a meus pais Genário e Lúcia que me deram tanto carinho e amor, sempre capazes de tudo por mim e por meus irmãos. Meu pai amado, que todos os dias lutou pela nossa educação e estava sempre disposto a sacrificar-se para nos dar tudo que entendia ser o melhor, me levava na garupa da bicicleta a noite contando histórias que me faziam viajar o mundo sem sair de lá. A minha amada mãe, doce e companheira de todas as horas que sempre me fez sentir o filho mais amado do mundo, mesmo com a distância, que perdeu muitas noites de sono pela pouca saúde da infância, uma mãe exemplar que junto com meu pai, são meus maiores exemplos de pessoa.

A meu irmão Igo, que é um companheiro para todas as horas, amigo de brincadeiras, de conversas longas sobre os assuntos da vida, de caçada pelas veredas do sertão que foi e será palco de nossas aventuras. A minha irmã Luana, menina forte, amiga e leal, das brincadeiras, dos sorrisos largos, da felicidade constante e de uma bondade cativante.

Agradeço a meus avôs, a minha avó Terezinha que ia comigo catar castanha de caju, fazia bolo de caco e sempre me mimava na infância. Ao meu avô Valdivino, que me levava pra aboiar o gado, me contava histórias da caipora e me levava pra feira dia de sábado.

Agradeço a minha namorada Camila, pela força, apoio e compreensão que tem demonstrado, pelo seu amor e companheirismo para comigo. Por sua contribuição decisiva neste trabalho, mostrando soluções quando eu mais precisei.

A meus tios e tias que deram uma contribuição importante nessa conquista. A meus amigos de infância, de escola de faculdade, que me mostraram o valor da amizade. Aos familiares e amigos que estão na paz eterna, que tento sonharam com esse momento, mas pelos caminhos da vida tiveram que partir.

A Rodrigo Teixeira, que co-orientou este trabalho, pelo seu apoio constante, por ter suportado a enxurrada de emails tendo respondido todos eles com indicação de caminhos que me davam animo e força a seguir neste trabalho.

A Augusto Sampaio, orientador deste trabalho, pela sua serenidade e seu senso de corretude. Sempre disponível, em todas as vezes que o procurei, seja por email, seja pessoalmente. Agradeço pela confiança que demonstrou desde a primeira vez que fui procurá-lo e pela atenção, leitura cuidadosa e conselhos que fizeram este trabalho melhor.

Agradecimento a Fernando Fonseca, tutor-paterno que sempre me tratou mais como um amigo que como um tutor, pessoa correta, me escutou e aconselhou em diversos assuntos acadêmicos e pessoais. A André Braga, amigo companheiro, que me ajudou em vários momentos de trabalho conjunto em prol dessa conquista.

Agradeço a todos os funcionários do CIn, pelo cafezinho, pelos banheiros limpos, pelas conversas nos momentos de aflição. Agradeço novamente a Deus por ter tantas pessoas a agradecer.

Resumo

Uma característica inerente a quase totalidade dos artefatos de software é a necessidade de evolução constante. À medida que o software é adaptado para se adequar a novos requisitos, este se torna mais complexo e passa a apresentar profundas diferenças entre código fonte e sua documentação, o que eleva os gastos com manutenção e torna o processo de desenvolvimento menos produtivo, além de dificultar o processo de integração dada à complexidade dos sistemas produzidos e sua inconformidade com a documentação disponível.

O desenvolvimento dirigido a modelos (MDD) apresenta na atividade de modelagem a chave para solução ou amenização dos problemas citados. Modelos formais são produzidos logo nas etapas iniciais do desenvolvimento e através de transformações são convertidos progressivamente em modelos menos abstratos e por fim em código fonte preservando em cada etapa a coerência entre os artefatos.

Uma das grandes demandas na implantação prática desse processo é a criação de ferramentas que automatizem a aplicação de transformações, o que envolve a busca por pontos no modelo passíveis de sofrerem uma dada transformação (casamento de padrões arquiteturais).

Este trabalho é uma contribuição no que diz respeito à automação tanto de *refactorings* em modelos quando da busca por padrões arquiteturais que permitam a aplicação destes. Um relevante subconjunto de *refactorinsg* (simples e compostos) para a linguagem de modelagem UML-RT foi automatizado utilizando o framework de metamodelagem Kermeta. A automação do casamento de padrões arquiteturais foi desenvolvida sob a forma de uma API, aplicável a qualquer linguagem definida como uma extensão Ecore. Mostramos como a partir de uma linguagem arbitrária *M*, um modelo válido *m* e um padrão *p* encontrar todos os pontos em *m* que casam com *p*. A implementação da API é apresentada em detalhes, constituindo um exemplo único da junção das linguagens OWL, SparQL, Java, Kermeta e Ecore e das APIs EODM e Protégé.

A automação das leis e a API de casamento de padrões arquiteturais possuem total integração entre si e com a com o framework de modelagem do Eclipse (EMF) [51].

Palavras-chave: casamento arquitetural, transformações de modelos, desenvolvimento dirigido a modelos, UML-RT, OWL, Kermeta.

Sumário

Lista de figuras

Lista de tabelas

1. Introdução

O desenvolvimento tradicional de software tem se mostrado incapaz de lidar de modo eficiente com a evolução constante que os artefatos de software estão sujeitos durante seu ciclo de vida [49]. Os artefatos produzidos nas fases iniciais de desenvolvimento, como documento de requisitos e digramas UML, tem seu conteúdo cada vez mais distanciado do estado atual do software à medida que avançam as etapas de codificação e testes [56]. Durante a etapa de manutenções, essa distância torna sem valor os documentos textuais e diagramas inicialmente produzidos.

A manutenção sem documentação acaba sendo feita totalmente via inspeção de código, o que torna o processo bastante improdutivo. Da mesma forma, a atualização constante dos documentos e diagramas torna o processo de desenvolvimento menos produtivo [49]. Além dos problemas com manutenção e produtividade, existem graves problemas acerca da interoperabilidade e dependência de plataforma, dado à vasta gama de novas tecnologias e plataformas de software que surgem cada vez mais rapidamente.

A abordagem de desenvolvimento orientada a modelos é uma resposta da academia a essa série de problemas que se apresenta ao mercado como uma alternativa ao desenvolvimento tradicional e seus artefatos [57]. MDA (*Model Driven Architecture*) é um framework de desenvolvimento de software, definido pela OMG (*Object Management Group*), que propõe a modelagem como atividade central na construção de software.

A atividade de modelagem começa com a elaboração de um modelo independente de computação (CIM), que após a fase de análise origina um modelo independente de plataforma (PIM). A fase de projeto, levando em consideração a tecnologia de implementação, utiliza o PIM para gerar um modelo específico de plataforma (PSM) que nas fases seguintes é transformado em código. A cada fase os modelos se tronam menos abstratos e mais relacionados com a tecnologia adotada.

A definição de MDA prevê que transformações entre os níveis de modelos, ou mesmo reestruturações em um dado nível, ocorram de forma automática preservando sempre a conformidade entre os vários modelos. Nessa abordagem, desenvolvedores focariam seus esforços na criação de modelos PIM que melhor suportassem as regras de negócio do sistema e através do uso de ferramentas de transformação seriam gerados, na seqüência, modelos PSM e código fonte.

Todo o esforço dos desenvolvedores concentrado na definição de um modelo PIM, gera ganho de produtividade e qualidade no sistema final, já que a complexidade da geração do PSM (e código fonte) torna-se responsabilidade das ferramentas de transformação. Um único PIM pode gerar PSMs para diferentes plataformas, aumentando assim a portabilidade do sistema. A manutenção, nessa abordagem, também é bastante favorecida, pois o PIM assume o papel de documentação de alto-nível do sistema e está sempre de acordo com o estado atual do mesmo, favorecendo a incorporação de novos requisitos através de mudanças no PIM seguidas de gerações automáticas de PSM e código [57].

Um tipo de transformação, previsto no framework MDA, é destinado a reestruturar o modelo mantendo-o no mesmo nível. Reestruturação (*refactoring*) é definida em [58] como sendo uma transformação de uma abstração em outra, de mesmo nível, preservando o comportamento externo do sistema (funcionalmente e semanticamente). A aplicação de *refactorings* tem como objetivo aumentar a qualidade do software através da inserção ou melhoria de requisitos não funcionais ou *soft-goals* (como, por exemplo, modularidade, manutenabilidade, reusabilidade) [32, 56]. A pouca disponibilidade de ferramentas que automatizem as transformações, peça central do framework MDA, é um entrave a sua utilização plena no mercado [57].

Este trabalho é uma contribuição à automação de um conjunto de reestruturações (*refactorings*) e construção de uma API de suporte ao casamento de padrões arquiteturais. Os *refactorings* implementados fazem parte de um considerável subconjunto de leis formalmente definido em [45] para modelos escritos em UML-RT [31] implementado utilizando o framework de metamodelagem Kermeta [13].

Modelos podem ser extensos e numerosos e a busca manual de locais onde se deve aplicar uma transformação é algo a ser claramente evitado, sob pena de perda de produtividade. A API desenvolvida neste trabalho tem o objetivo geral de auxiliar na busca dos locais de aplicação de qualquer lei de reestruturação, definida para linguagens estendidas de Ecore, no ambiente de modelagem EMF (Eclipse) [51].

Outras contribuições deste trabalho são a definição de uma arquitetura para a implementação de leis compotas [56], a revisão do metamodelo para UML-RT proposto em [59] e a utilização de uma coleção de linguagens e APIs de forma inédita na solução do problema de casamento de padrões arquiteturais.

No capítulo 2 deste trabalho apresentamos os conceitos básicos de MDA. Descrevemos as características e relacionamentos entre os modelos CIM, PIM e PSM, a maneira pela qual se define formalmente a semântica e estrutura de modelos e como estes podem ser reestruturados ou transformados em modelos com diferentes níveis de abstração através de transformações de modelos.

No capítulo 3 apresentamos a estrutura da implementação das transformações (*refactorings*) abordadas neste trabalho. Iniciamos com a descrição da linguagem UML-RT e das linguagens/frameworks disponíveis para implementação das leis, destacando as linguagens MOLA [9] , QVT [20] e Kermeta, utilizadas na implementação deste trabalho. Detalhamos o metamodelo proposto para UML-RT [59], bem como a linguagem XMI [60] em que modelos válidos são representados. Ao final apresentamos os diagramas de classe para implementação de leis simples e compostas bem como exemplos de sua aplicação, destacando a forma de leitura, validação, transformação e persistência dos modelos.

O capitulo 4 é dedicado à descrição da API de casamento de padrões arquiteturais implementada neste trabalho. Iniciamos com uma breve descrição a cerca da definição formal de padrões arquiteturais e dos objetivos a serem alcançados com a API. Apresentamos um estudo a cerca de duas linguagens de representação de conhecimento, FLORA-2 [44] e OWL (e sua linguagem de consulta SparQL) [52,53], explicada em maiores detalhes, dado que esta foi a linguagem escolhida para representação do conhecimento presente nos modelos (OWL) e execução de consultas com base nos

padrões(SparQL). Descrevemos o diagrama de classes para da API e a forma pela qual interage com as APIs EODM [54] e Protégé [55] para prover o serviço de busca de padrões. Ao final mostramos um exemplo de uso da API na busca de um padrão associado a uma transformação seguida da aplicação da mesma.

No capítulo 5 apresentamos as conclusões deste trabalho e discutimos alguns trabalhos relacionados e futuros.

2. Arquitetura Dirigida a Modelos

Na última década tem se difundido uma abordagem de desenvolvimento de software que tem na atividade de modelagem a principal forma de conceber, construir e manter software. O desenvolvimento dirigido a modelos (MDD) [61] faz parte de um conjunto de respostas que a indústria de software tem proposto aos números que indicam a pouca produtividade, qualidade e confiabilidade da mesma se comparada a outras indústrias que alcançaram um maior grau de maturidade [49].

O desenvolvimento dirigido a modelos foca na definição de modelos que capturem corretamente a semântica do ambiente onde reside o problema que se quer abordar, seguida da transformação destes modelos em outros que estão em um nível de abstração menor e, portanto mais próximos da plataforma específica, onde a realização desses modelos irá ser implantada sob a forma de um componente executável.

Arquitetura dirigida a modelos (MDA) [49] é uma abordagem de desenvolvimento, produção e manutenção de software, desenvolvida e mantida pela OMG™ [62] que em linhas gerais é uma implementação/visão da abordagem MDD. MDA não se relaciona com qualquer plataforma ou software específico sendo, sobretudo, um framework conceitual extensível que descreve uma abordagem geral para o desenvolvimento de software.

A OMG tem definido os conceitos gerais de MDA e tem formado uma serie de grupos de trabalho para definir os padrões necessários a realização desses conceitos. Uma lista de padrões largamente usados e de notória estabilidade como: CORBA (C*ommon Object Request Broker Architecture*) [63], UML (*Unified Modeling Language*) [64], MOF (*MetaObject Facility*) [65] e XMI (XML *Metadata Interchange*) [22] permitem a concretização das diretrizes propostas pela abordagem MDA.

A figura abaixo ilustra a idéia síntese de MDA: expressar o software através dos padrões básicos definidos pela OMG (e.g. UML, MOF, CWM) e, através de regras de transformações, gerar representações compatíveis com as plataformas que darão suporte a execução do software (e.g,Java,.NET).

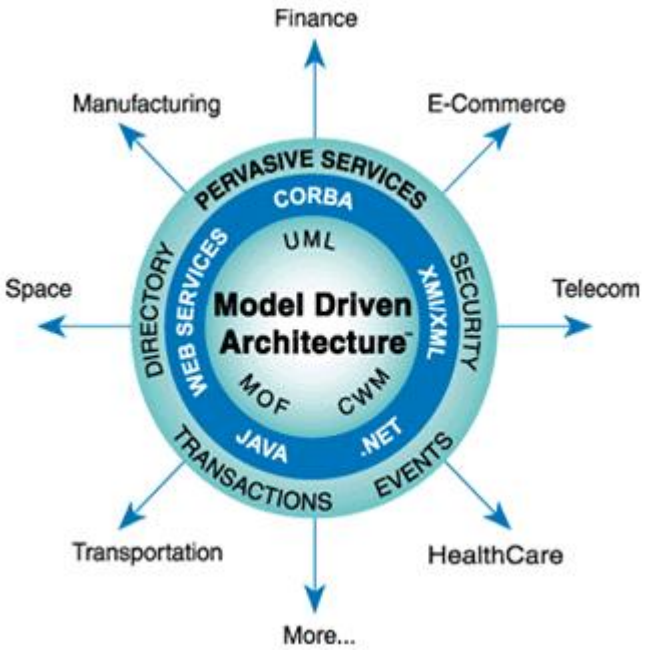


Figura 2.1 - Visão MDA da OMG

Alguns conceitos base precisam ser formalizados para um completo entendimento acerca de MDA. São eles: modelos, semântica do domínio (metamodelos) e transformações.

2.1 Modelos: CIM, PIM e PSM

Modelos, no sentido amplo da palavra, são abstrações de algo que existe ou que se deseja construir. Essa definição pode ser contextualizada na área de engenharia de software, sob o foco de MDA, e assim passamos a dizer que um modelo é a especificação formal da estrutura e do comportamento de um sistema [49].

Em MDA existe três níveis de abstração de modelos: independentes de computação (CIM), independentes de plataforma (PIM) e específicos de plataforma (PSM).

2.1.1 Modelos Independentes de Computação

São aqueles referentes ao modelo de negocio que a solução de software ira se integrar. Usa uma notação compreensível pelos especialistas do domínio. Tem todas as informações do que o software deve fazer, mas nenhuma referência a qualquer informação sobre tecnologia ou computação. São utilizados principalmente na definição de requisitos do sistema [66]. Funcionam como uma comunicação entre os especialistas do domínio e os profissionais de tecnologia da informação.

2.1.2 Modelos Independentes de Plataforma

Estes modelos definem um conjunto de serviços que o software deverá prover, sem se ater aos detalhes de nenhuma plataforma de software específica. Estes modelos representam o núcleo funcional e estrutural do sistema.

A diferença síntese entre CMI e PIM é que este já apresenta uma visão computacional do sistema (sem ligação a nenhuma plataforma), ausente no primeiro.

2.1.3 Modelos Específicos de Plataforma

Dado um PIM e um conjunto de informações sobre uma dada plataforma (e.g, Java, .NET) é possível gerar um modelo específico de plataforma onde irá residir o executável do sistema. Um PIM pode gerar mais de um PSM, em caso de sistemas multiplataformas, o que bastante comum nos dias atuais. Isso traz grande redução de custo com recodificação e faz o desenvolvedor focar sua atenção no PIM, ou seja, nas funcionalidades e lógica de negócio do sistema.

2.2 Formalização de Modelos

Para que qualquer modelo possa ser lido, interpretado, alterado e persistido computacionalmente é necessário que este tenha sido escrito em uma linguagem bem definida (possui sintaxe (forma) e semântica (significado) bem definidas).

A definição de uma linguagem que possa ser tomada como base para e escrita de modelos é feita através do mecanismo de metamodelagem. Um metamodelo define formalmente a linguagem em que um modelo será escrito [66, 49]. Um metamodelo é também um modelo e, portanto precisa ser escrito em uma linguagem bem definida. Esta por sua vez é definida através de um metametamodelo. A figura abaixo ilustra essa seqüência.

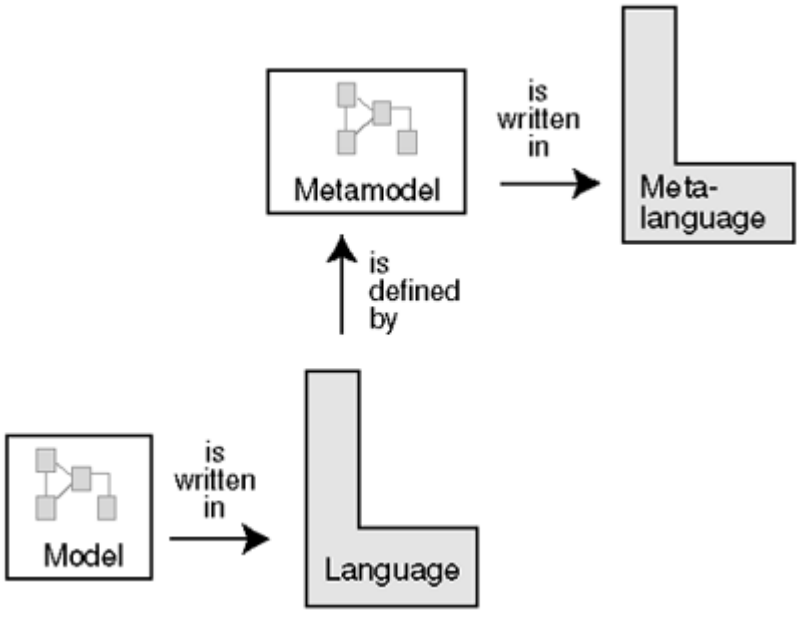


Figura 2.2 - Relação entre modelos e metamodelos

Essa seqüência pode prosseguir indefinidamente, porém em MDA somente são adotados quatro níveis:

2.2.1 Nível M0: Instâncias

Neste nível estão representadas as instâncias das abstrações que, num dado instante, estão sendo manipuladas pelo sistema. O conjunto dessas instâncias é dinâmico em tempo de execução. Podemos entender as instâncias como sendo a realização ou concretização das abstrações definidas no modelo, que podem estar no banco de dados ou na memória de trabalho do computador.

2.2.2 Nível M1: Modelo

Descreve as abstrações do domínio do sistema (forma e significado). Os conceitos presentes em M1 são categorizações ou classificações para as instancias de M0. Um exemplo simples é mostrado na figura abaixo. Um modelo em UML descreve algumas abstrações de um sistema de vendas e estas são concretizadas com a criação de três instâncias.

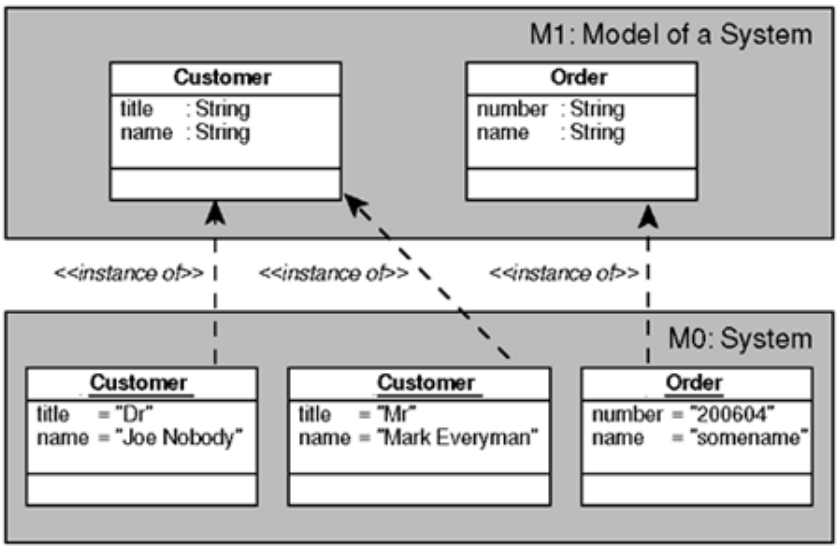


Figura 2.3 - Relação entre M0 e Mi

2.2.3 Nível M2: Metamodelo

Assim como o modelo no nível M1 define as instâncias que podem estar presentes em um modelo válido M0, o metamodelo M2 define que instâncias estão presentes e como elas podem se relacionar, de modo válido, no modelo M1. No exemplo mostrado na Figura 2.4 as classes *Costumer* e *Order* representam instâncias do metamodelo M2. Elementos do modelo UML em M1 são instâncias do metamodelo M2 que descreve forma e significado de um modelo válido em UML.

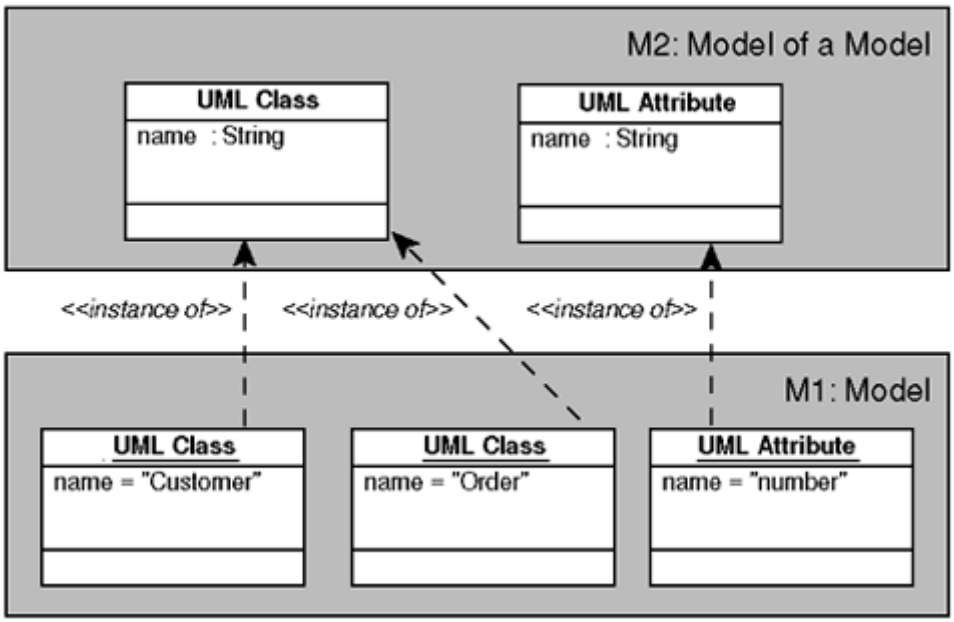


Figura 2.4 - Relação M2 com M1

2.2.4 Nível M3: Metametamodelo

Semelhantemente aos demais conceitos, metametamodelos em M3 definem a semântica e estrutura de metamodelos na camada M2. MDA adota M3 como último nível de abstração de modelos. Uma pergunta simples que surge é: se este é o último nível, como se define a sintaxe e semântica de modelos M3? A resposta a essa questão em MDA é que metametamodelos definem seu próprio significado e estrutura, sendo assim o último nível necessário [61].

2.3 Transformações de modelos

Transformações em modelos constituem um ponto crucial na abordagem MDA. A entrada de qualquer transformação é um modelo e a saída pode ser outro modelo ou código executável em alguma plataforma (e.g. Java, .NET). Existem três categorias de transformações: *refactorings*, modelos para modelos, modelos para código.

I. *Refactorings*: Dado um modelo de entrada (CIM, PIM ou PSM) esta transformação gera como saída um modelo revisado, reorganizado segundo algum critério. Esse modelo de saída refatorado preserva a mesma semântica do modelo de entrada, porém com uma estrutura diferente [56]. Um exemplo simples de *refactoring*, em um modelo UML, pode ser o estabelecimento de uma relação entre duas classes, ou a mudança do nome de um dado atributo. Uma condição necessária para que uma transformação seja considerada um *refactoring* é que qualquer mudança no modelo não será percebida por qualquer entidade externa que use os serviços disponibilizados pelos elementos presentes no modelo.

II. Modelos para Modelos: esta transformação geralmente é aplicada na geração de um modelo em um nível de abstração menor que o modelo de entrada. Dado um modelo e um conjunto de informações adicionais gera-se um modelo de saída em um nível de abstração inferior.

III. Modelos para código: esta transformação gera a código ou trechos de código executável a partir de um dado modelo. Algumas ferramentas geram a partir de um modelo UML código Java ou C#.

A figura abaixo mostra uma visão geral de como as transformações interagem no fluxo de desenvolvimento MDA:

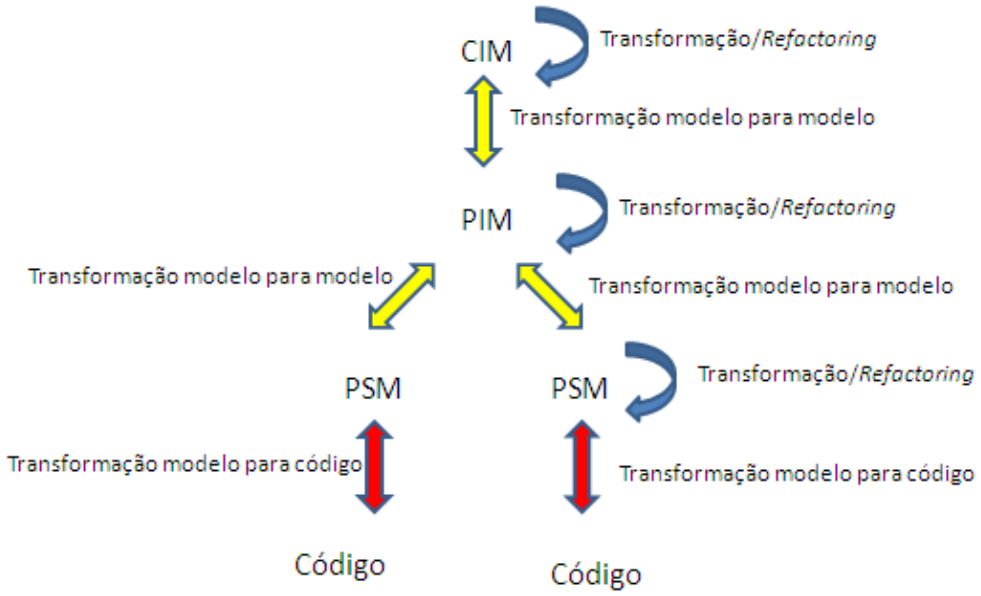


Figura 2.5 - Transformações de modelos no fluxo de desenvolvimento MDA

Na Figura 2.5 as transformações entre CIM e PIM e PIM e PSM são tipicamente transformações modelos para modelos; as transformações entre PSM e código são transformações modelo para código; as transformações que mantém o modelo no mesmo nível de abstração são *refactorings*, ou reorganizações dos modelos.

Outra classificação para *refactorings* se baseia na forma como eles são aplicados, eles podem ser aplicados de forma totalmente manual, semi-manual (com algum auxílio ferramental) ou completamente automatizados. Um dos focos centrais deste trabalho está em automatizar um considerável conjunto de *refactorings* em modelos de nível de abstração PIM.

3 Implementação de Transformações

As transformação implementadas nesse trabalho formam um considerável conjunto das leis de *refactoring* propostas para modelos escritos na linguagem de modelagem UML-RT [31] formalmente definidas em [45].

3.1 UML-RT

A linguagem de modelagem UML-RT é uma extensão conservativa de UML que é focada na definição de sistemas baseados em componentes e sistemas embarcados brandos, que são sistemas que possuem um conjunto de tarefas com restrições temporais associadas, onde o não cumprimento de uma data restrição traz apenas um decréscimo de desempenho.

UML-RT mantém todos os elementos já existentes em UML com mesma semântica e sintaxe, adicionando novos elementos como portas, protocolos, cápsulas e máquinas de estado, como elementos que darão suporte a definição de componentes ativos e suas relações estruturais e comportamentais. Esses elementos são heranças da linguagem ROOM [67], que influenciou a definição de UML-RT.

Cápsulas são elementos ativos que representam um fluxo de controle independente do sistema. Cápsulas, assim como classes, podem ter atributos e métodos, além estabelecer relacionamentos de dependência e generalização, sendo este último um tópico ainda pouco claro na comunidade científica. Existe uma série de diferenças entre cápsulas e classes:

a. A comunicação entre classes se dá através da chamada de métodos definidos nas classes, ao passo que em cápsulas, a comunicação se dá exclusivamente pelo envio de mensagens. Uma cápsula pode, a qualquer momento, enviar uma mensagem através de uma de suas portas para uma porta (a qual esteja ligada) de outra cápsula, que eventualmente pode enviar alguma mensagem de retorno.

b. Cápsulas podem invocar métodos de classes, mas o inverso não se aplica.

c. Todos os atributos e métodos de uma cápsula são necessariamente ocultos ao meio exterior, com exceção de suas portas, que podem ser visíveis ou não ao ambiente externo a cápsula.

Cada porta de uma cápsula recebe/envia mensagem que obedecem a um determinado formato, este formato é definido por um protocolo. Protocolo é um contrato que define o formato das mensagens válidas que podem ser trocadas entre as duas portas que participam do protocolo.

O comportamento de uma cápsula é acionado quando esta recebe uma mensagem em alguma de suas portas, nesse momento, algumas operações são necessárias para construir a mensagem de retorno, ou simplesmente efetuar o processamento requisitado. Para isso, uma máquina de estado é associada à cápsula, para representar seu comportamento. A máquina de estado é o único elemento que pode acessar as partes não visíveis externamente de uma cápsula.

Um exemplo de modelo escrito em UML-RT é mostrado na figura abaixo:

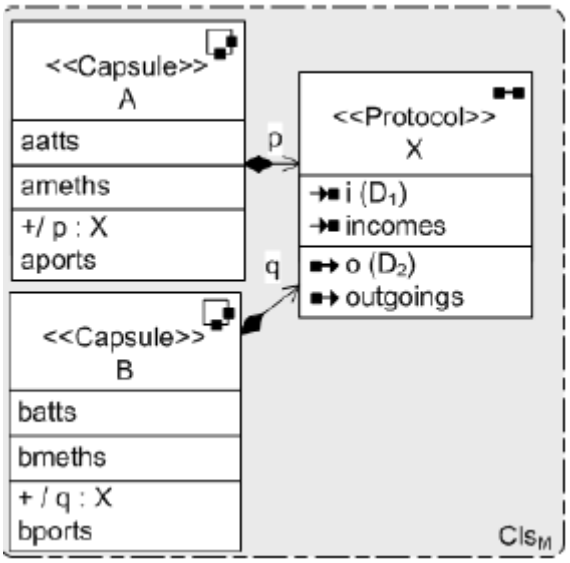


Figura 3.1 - Modelo em UML-RT

Na Figura 3.1 são representadas duas cápsulas, *A* e *B* com seus atributos e métodos além de duas portas públicas *p* e *q*, estas portas estão conectadas por um protocolo *X* que define os sinais de entrada e saída que compõe as mensagens trocadas entre as portas.

3.2 Tecnologias

Dada a definição de uma linguagem em que serão representados os modelos a serem transformados/refatorados existe uma vasta gama de frameworks/linguagens que permitem a definição e aplicação de transformações: VIATRA (*VIsual Automated model TRAnsformations*) [1], Kent Model Transformation language [2], Tefkat [3], GReAT (*Graph Rewriting and Transformation language*) [4], ATL (*Atlas Transformation Language*) [5], UMLX [6], AToM3 (*A Tool for Multi-formalism and Meta-Modeling*) [7], BOTL (*Bidirectional Object-oriented Transformation Language*) [8], MOLA (*MOdel transformation LAnguage*) [9], AGG (*Attributed Graph Grammar system*) [10], AMW (*Atlas ModelWeaver*) [11], triple-graph grammars [12], MTL (*Model Transformation Language*) [13], YATL (*Yet Another Transformation Language*) [14], Kermeta [15], C-SAW (*Constraint-Specification Aspect Weaver*) [16] e XSLT [18]. Algumas delas são representativas de algumas categorias de agrupamento das linguagens de transformação e são detalhadas abaixo:

3.2.1 MOLA

MOLA (*MOdel Transformation LAnguage*) é uma linguagem de transformação gráfica que possui alguns elementos textuais. A sintaxe abstrata para os elementos gráficos é definida via metamodelo e através de uma BNF para os elementos textuais.

Uma definição de uma transformação MOLA consiste de um metamodelo e um conjunto de procedimentos. O metamodelo descreve o modelo a ser transformado

e o procedimento descreve o algoritmo de transformação. Tanto o metamodelo como procedimentos, que definem uma transformação, são diagramas gráficos (com alguns elementos textuais) e, portanto são ambos descritos usando o metamodelo de MOLA.

As figuras abaixo mostram, na seqüência, um exemplo simples de uma regra/procedimento em MOLA, um modelo de entrada para a regra e modelo de saída após sua aplicação:

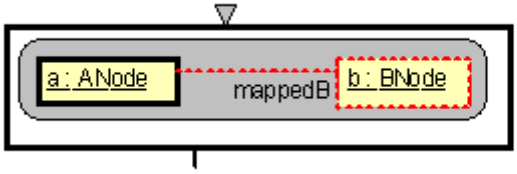


Figura 3.2 - Regra gráfica MOLA

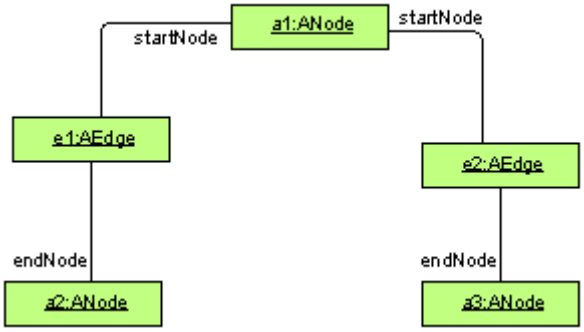


Figura 3.3 - Modelo de entrada da transformação

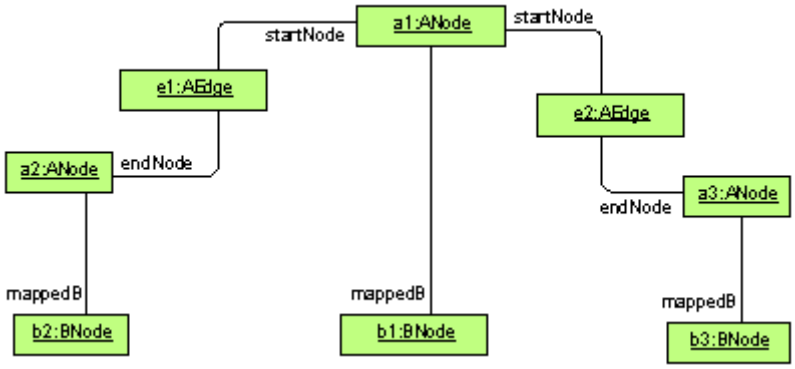


Figura 3.4 - Modelo de após transformação

Uma regra em MOLA define um padrão e uma ação associada. O padrão da Figura 3.2 casa com todas as instâncias em um modelo do tipo *ANode* e a ação é que, pra cada uma delas, se adiciona uma instância de *BNode* e um link de mapeamento do tipo *mappedB*. No modelo da Figura 3.3 - Modelo de entrada da transformação, três

instâncias de *ANode* são encontradas e para cada uma delas a regra é aplicada gerando o modelo da Figura 3.4. Assim para as instâncias de *ANode* (*a1*, *a2* e *a3*) são criadas as instâncias de *BNode* (*b1*,*b2* e *b3*) e entre cada uma deles se estabelece a relação *mappedB* (a_i,b_i), onde i Ɛ{1,2,3}.

Uma das principais desvantagens de MOLA é ausência de uma abordagem, em nível de linguagem, para a composição de transformações. Outro considerável problema é que transformações bidirecionais ou incrementais têm que ser construídas manualmente já que MOLA é uma linguagem deliberadamente procedural.

3.2.2 QVT

QVT (*Query/View/Transform*) é o padrão definido pela OMG para transformação de modelos. A especificação de QVT é de natureza dual, possuindo uma parte declarativa e outra imperativa. A parte declarativa está estruturada em duas partes, *Relations* e *Core*, enquanto a parte imperativa é dividida em *Operational Mappings* e *Black Box* como mostrado na figura abaixo:

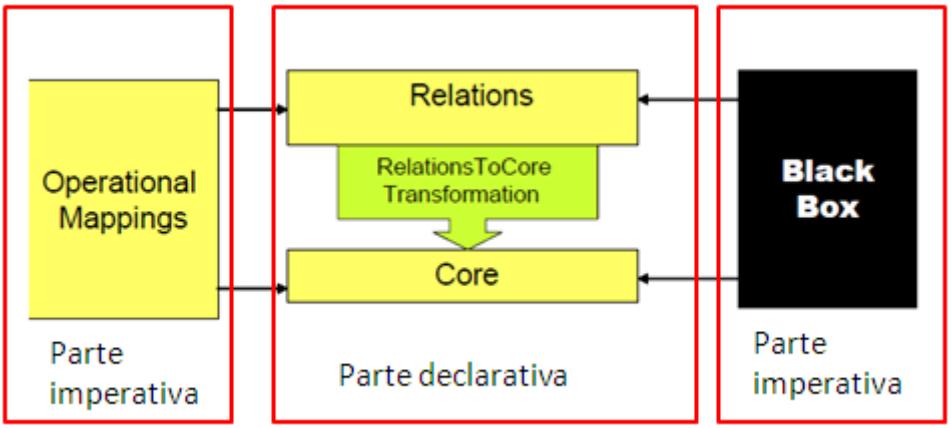


Figura 3.5 - Relação entre os metamodelos de QVT

A linguagem *Relations* define uma especificação declarativa das relações entre objetos MOF (*Meta Object Facility*). A linguagem suporta casamento de padrões em objetos complexos, e cria implicitamente estruturas que armazenam o que ocorreu durante a execução de uma transformação. Possui um maior nível de abstração que a linguagem *Core*, sendo as duas equivalentes e conversíveis entre si.

Core é uma pequena linguagem, com sintaxe simplificada, que permite somente casamento de padrões sobre um conjunto plano de variáveis. *Core* é igualmente poderosa à linguagem *Relations*, e por ser menos complexa em suas construções, é definida de forma bastante simples. O fato de ser uma linguagem de construções simples faz com que as transformações descritas em *Core* sejam excessivamente extensas, se comparadas com as equivalentes em *Relations.* Outra diferença, entre as duas partes declarativas, é que as estruturas que armazenam o que ocorreu durante a aplicação da transformação têm que ser definidas explicitamente em *Core*.

Operational Mappings está para a parte imperativa assim como *Relations* está para a parte declarativa. A linguagem provê extensões de OCL com efeito colateral, o que permite um estilo procedural de programação. Esta pode ser usada para implementar uma transformação em conjunto com *Relations* quando esta é de complicada representação utilizando somente o paradigma declarativo. Vale salientar que transformações podem ser completamente definidas em cada uma das linguagens ou como uma combinação das duas.

Uma ou mais implementações *Black Box* podem se plugar a com as linguagens *Relations* ou *Core*, desde que respeitem a interface de operações MOF, que define as operações válidas sobre os modelos baseados em MOF. Este acoplamento permite:

- Que algoritmos complexos codificados em alguma outra linguagem de programação que possua *binding* com MOF sejam utilizados (por exemplo, XSLT [18], XQuery [68]),
- Utilização de bibliotecas específicas para calcular valores de propriedades dos modelos e
- Transparência na implementação de algumas partes das transformações.

Existe uma série de implementações para a especificação QVT. A maioria das implementações aborda somente uma das partes da especificação, o que impede que todos os benefícios da especificação possam ser alcançados. SmartQVT [19] apresenta uma implementação para a parte operacional de QVT (*Operational Mappings*) possuindo integração com a plataforma Eclipse[21]. Medini QVT [20] é uma das implementações para a linguagem *Relations* que tem sido bastante utilizada por sua fácil integração com Eclipse (editor textual assistente de transformações, *debugger*, *trace* das transformações) e alta adesão com a especificação QVT.

Embora exista uma série de implementações para cada parte da especificação QVT, um dos problemas principais continua sendo o de integração, o que dificulta a adoção plena em soluções MDA.

3.2.3 Kermeta

Kermeta é uma DSL (*Domain Specific Language*) definida para engenharia de metamodelos. Ela preenche a lacuna deixada pela especificação MOF, que somente define a estrutura de metamodelos, adicionando uma forma de especificar semântica estática (similar a OCL) e semântica dinâmica (usando semântica operacional nas operações do metamodelo), definindo assim, tanto a estrutura quanto o comportamento dos modelos. Kermeta é compatível com a especificação EMOF (parte núcleo da especificação MOF 2.0 da OMG) [65] e o Ecore (plataforma Eclipse) [51].

O intuito da linguagem é se tornar a linguagem núcleo de uma plataforma orientada a modelos, por isso, ela pode ser encarada como uma linguagem base para implementar linguagens de matadados, de ações, de *constraints* ou de transformações.

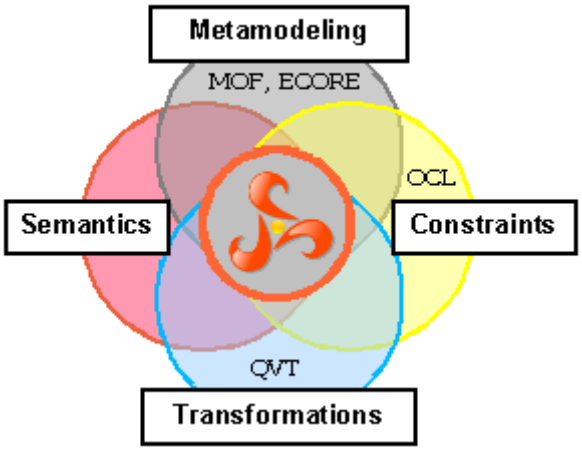


Figura 3.6 - Posicionamento de Kermeta

Na Figura 3.6 Kermeta aparece como sendo uma intersecção de características de outras linguagens, permitindo: definição de *contraints* (OCL), transformações de modelos (QVT), ambiente de metamodelagem (compatível com MOF, Ecore) e definição de semântica para os modelos, que como dito anteriormente, veio a preencher a lacuna deixada pela especificação MOF da OMG.

Kermeta é uma linguagem orientada a modelos, possuindo uma série de facilidades para ler, persistir, transformar e exportar modelos. Possui características de linguagem imperativa e orientada a objetos.

A sintaxe imperativa de Kermeta é baseada em Eiffel [69]. O código é estaticamente tipado e checado, e a execução é feita por um interpretador. Possui todos os *statements* usuais como blocos, condições de controle e iteração.

```
var v1 : Integer init 2
var v2 : String init "blah"

if v1 > 5 then v1 := v1-5

if v1 == 2 then
    v2 := v1
    v1 := v2 + v1
else
    v1 := 0
end
```

```
var v1 : Integer init 3
var v2 : Integer init 6

from var i : Integer init 0
until i == 10
loop
    i := i + 1
end
```

Figura 3.7 - Estruturas de controle e iteração em Kermeta

A Figura 3.7 mostra no lado esquerdo uma estrutura aninhada de controle. A palavra reservada *end* marca o termino de um bloco ou função. À direita da figura uma estrutura de iteração tem inicio com a palavra reservada *from*, condição de parada em *until*, e o código a ser executado entre *loop* e *end*.

Kermeta provê amplo suporte ao paradigma OO, permitindo definição de classes, herança simples, atributos e operações. É possível expressar sintaticamente noções de atributos (relações de composição) e associações distinguindo

explicitamente esses dois conceitos, diferindo assim das linguagens OO de propósito geral como Java ou C#.

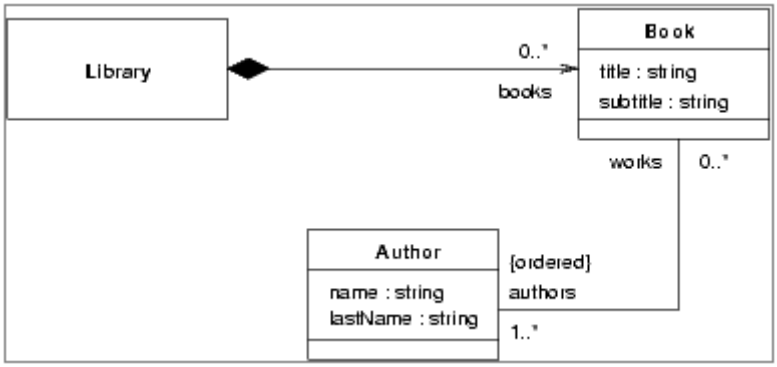


Figura 3.8 - Digrama UML simplificado de uma biblioteca

```
class Library
{
    attribute books : set Book[0..*]
}

class Book
{
    attribute title : String
    attribute subtitle : String

    reference authors : oset Author[1..*]#works
}


class Author
{
    attribute name : String
    attribute lastName : String

    reference works : set Books[0..*]#authors
}
```

Figura 3.9 - Representação em Kermeta do modelo da biblioteca

A Figura 3.8 mostra um modelo simplificado de uma biblioteca. A Figura 3.9 - Representação em Kermeta do modelo da biblioteca exibe a representação do modelo em Kermeta. A palavra reservada *attribute* define uma relação de composição com o conteúdo (*title* e *subtitle* em *Book*), enquanto *reference* define uma relação de associação simples (*authors* e *works* em *Book* e *Author*, respectivamente). O tipo *set Books* indica um conjunto não ordenado de referências para o tipo *Book*, já *oset Author* indica um conjunto ordenado de referências para o tipo *Author*.

Escolhemos Kermeta para implementar as transformações descritas em [45] pelos seguintes motivos:

- Compatibilidade com o padrão MOF da OMG que é largamente difundido
- Fácil definição de *constraints* ao estilo OCL para verificar pré/pós-condições no modelo
- Integração com a plataforma Eclipse e como seu metamodelo Ecore
- Facilidade de integração com a linguagem Java, o que foi imprescindível para a implementação do casamento arquitetural semântico e a integração deste com as transformações implementadas.

3.3 Metamodelo UML-RT

O primeiro passo para implementar as transformações abordadas neste trabalho foi a definição de um metamodelo que capturasse a estrutura e significado de UML-RT. O metamodelo apresentado aqui é uma melhoria em relação ao proposto em [59], através da remoção de elementos mais ligados à implementação das leis que ao significado de UML-RT, e pela adição de outros com objetivo de aumentar a granularidade das estruturas, condição necessária para a realização de algumas transformações mais elaboradas.

O metamodelo apresentado neste trabalho foi definido como uma extensão do metamodelo Ecore que é compatível com MOF e Kermeta. Pela equivalência entre Kermeta e Ecore o metamodelo de UML-RT foi automaticamente transformado em sua representação em linguagem Kermeta.

A seqüência de figuras abaixo mostra o metamodelo proposto para UML-RT:

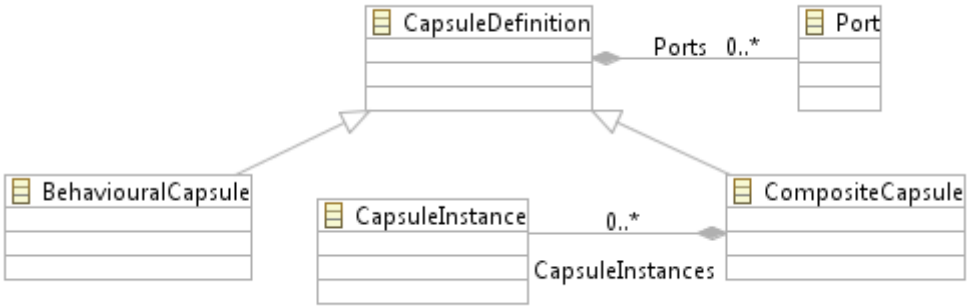


Figura 3.10 - Tipos de Cápsula

Na figura acima *CapsuleDefinition* representa a definição de uma cápsula genérica que possui um conjunto vazio ou não de portas(elementos do tipo *Port)*. Uma cápsula do tipo *BehaviouralCapsule*, além de portas, possui uma máquina de estado, que como foi visto, representa o comportamento de uma cápsula. Por fim *CompositeCapsule* é uma cápsula que possui um conjunto de *CapsuleInstance*, que são elementos estruturais do modelo, representando uma instância de algum subtipo de *CapsuleDefinition.*

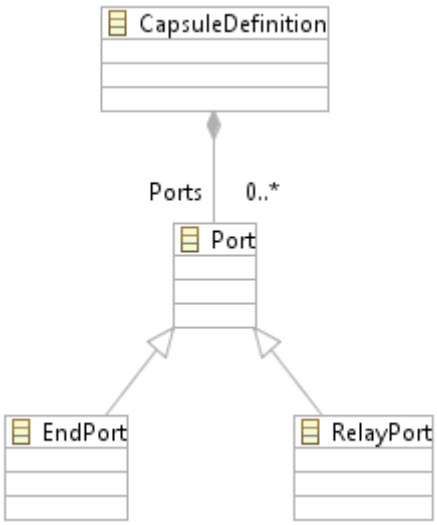


Figura 3.11 - Tipos de porta

Como mostrado na figura acima há dois tipos de porta: *EndPort* e *Relayport*. Portas *relay* são sempre públicas e ligadas a outras portas. São usadas para enviar sinais de eventos diretamente a componentes privados da cápsula, sem serem processados pela própria cápsula. Geralmente são usadas para expor as interfaces das instâncias de uma cápsula composta (*CompositeCapsule*). Portas *end* podem ser públicas ou privadas, ligadas ou não a outras portas. Podem ser usadas para enviar sinais para serem processados diretamente pela máquina de estados de uma cápsula. Portas *end* são o último destino de todos os sinais enviados por uma cápsula. Estes sinais são gerados em máquinas de estado e recebidos por máquinas de estado.

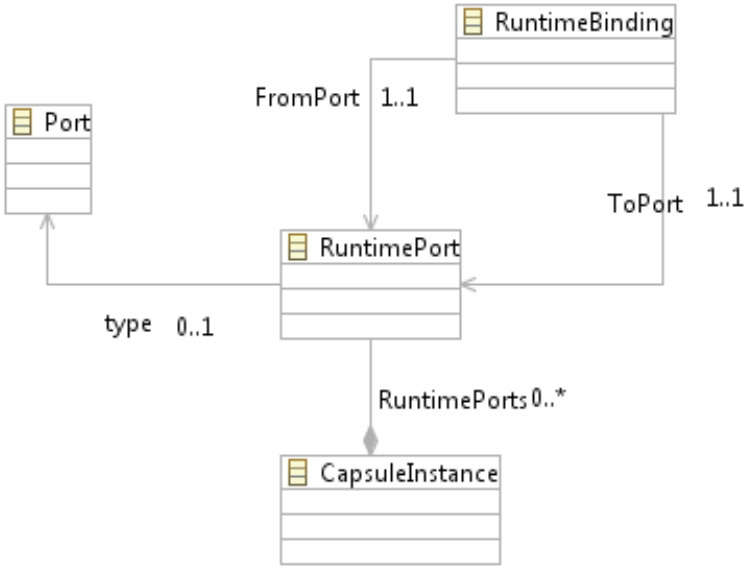


Figura 3.12 - Ligações entre portas

Uma instância de cápsula (*CapsuleInstance*) tem um conjunto de *RuntimePort*, que representa um ponto de junção. Uma ligação entre dois pontos é representado por *RuntimeBinding*, que tem um ponto de origem (*FromPort*) e o de destino (*ToPort*). Embora essa ligação seja unidirecional, os sinais enviados entre os pontos de junção podem trafegar em ambas as direções (ida/volta).

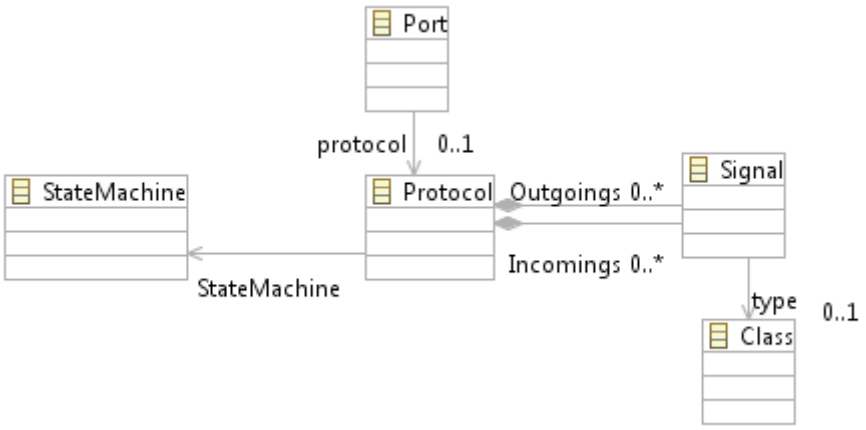


Figura 3.13 - Portas e protocolos

A toda porta está associado um protocolo (*Protocol*), que define um conjunto de sinais de entrada (*Incomings*) e saída (*Outgoings*) para aquela porta. Um protocolo pode possuir uma máquina de estado caso um de seus sinais necessite de algum processamento antes de seguir adiante.

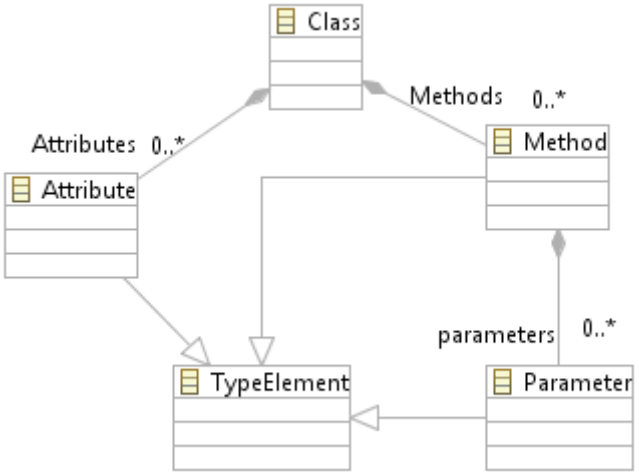


Figura 3.14 - Definição de classe

Uma *Class* representa um objeto passivo, conceito mantido de UML. Possui um conjunto de *Attributes* e *Methods*. O elemento *Attribute* é um subtipo de *TypeElement* assim como *Parameter*. Um método tem um conjunto de parâmetros (*parameters*) e é também um subtipo de *TypeElement*. De fato numa expressão o retorno do método é representado como sendo seu próprio tipo.

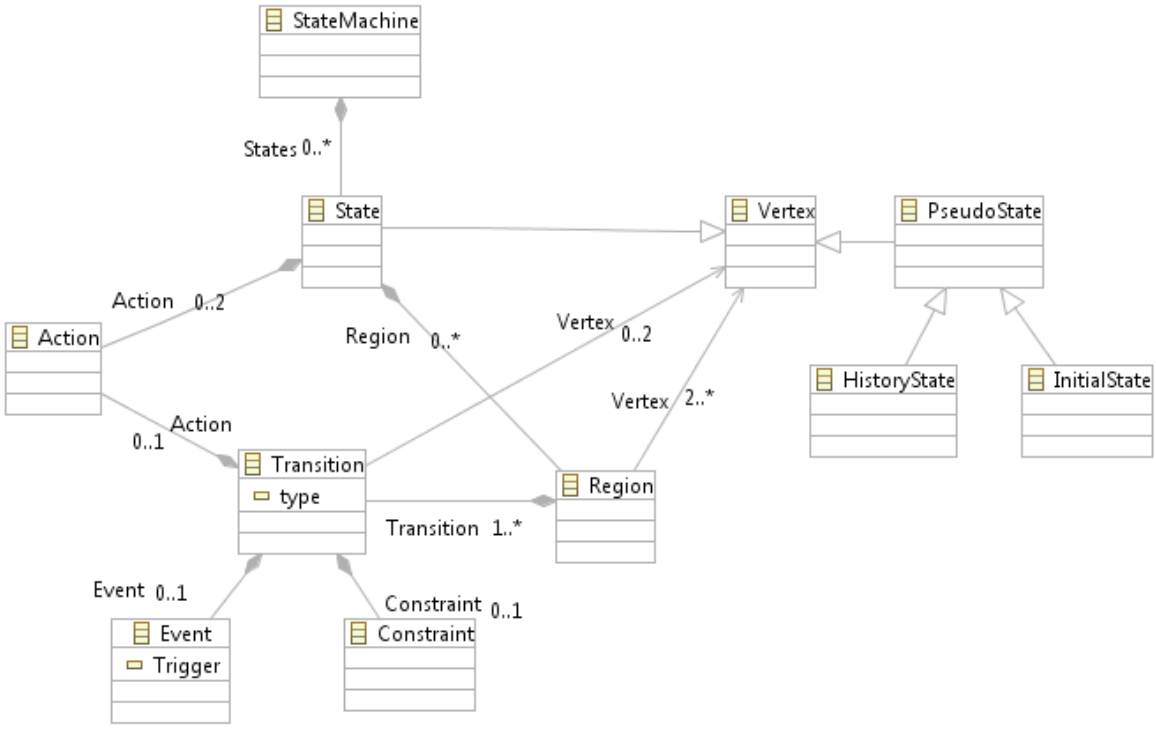


Figura 3.15 - Máquina de Estados

Tanto protocolos quanto cápsulas têm uma máquina de estados, que representa seu comportamento. *StateMachine* representa máquinas de estados. Possui um conjunto de estados *State*, subtipo de *Vertex* ,que representa um estado genérico. Um *State* tem duas ações associadas (*Action*), uma que é executada ao se entrar no estado e outra que executa na saída do mesmo. Um estado pode ser simples ou pode ser composto de outros subestados, portanto para os estados compostos há a necessidade de se guardar outros estados e as transições entre eles, para isso um estado, quando composto, contém um conjunto de regiões *Region*.

Cada *Region* possui um conjunto de vértices *Vertex* e todas as transições entre os mesmos (conjunto de *Transition*). Uma *Transition* representa uma ligação unidirecional entre dois estados, representada pela relação *vertex*. Cada transição, ao ser efetuada, executa uma ação dada por *Action* e é disparada na ocorrência de um ou mais eventos (*Event*). Um *PseudoEstate* é um tipo de *Vertex* que pode representar o estado inicial da *StateMachine*, ou um estado que guarda o trace de execução da máquina.

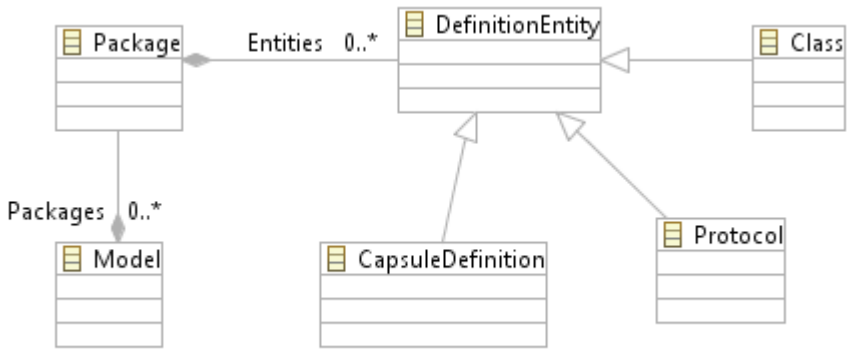


Figura 3.16 - Pacotes e seus elementos

Por fim, um modelo *Model* é constituído por um conjunto de pacotes, *Package*. Um *Package* possui um conjunto de entidades, que podem ser definições de cápsulas, protocolos ou classes.

3.4 Estrutura das Transformações

Dada a definição do metamodelo, o próximo passo foi a elaboração de uma estratégia de implementação das transformações, composta por: escolha do conjunto de transformações a serem implementadas, transformação do metamodelo definido para a linguagem Kermeta, escolha da linguagem de representação dos modelos, elaboração da arquitetura das transformações (simples e compostas) e por fim validação dos resultados obtidos.

3.4.1 Transformações Abordadas

Para implementação foram escolhidas as leis mostradas no quadro abaixo:

Declarar Cápsula	Introduzir Associação Cápsula-Cápsula
Introduzir Método/Atributo	Introduzir Associação Cápsula-Classe
Introduzir Associação Cápsula-Protocolo	Introduzir Conexão
Substituição de Cápsula	Encapsular Cápsula
Compor Protocolos	Reescrever Ação de Transição
Criar região	Isolar Ação
Extrair Classe	**Decomposição Paralela de Uma Cápsula**

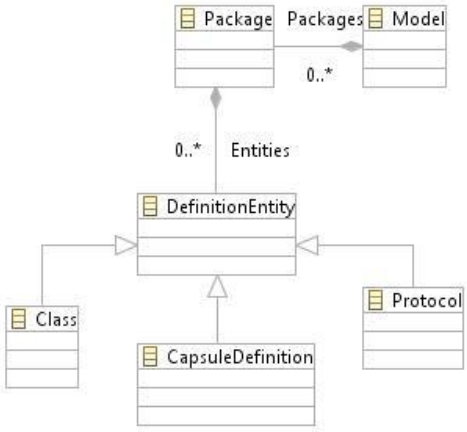
Tabela 3.1- Leis implementadas

As leis mostradas na Tabela 3.1 podem ser dividas entre leis simples e compostas (em negrito). Leis compostas são obtidas como composição de um conjunto de leis simples. Cada lei é formada por um *template* representando o lado esquerdo e um *template* representante o lado direito da transformação. Todas são bidirecionais e possuem em cada sentido da aplicação um conjunto de pré-condições.

3.4.2 Transformação do metamodelo Ecore para Kermeta

Dado o metamodelo definido na plataforma Eclipse como uma extensão de Ecore, este foi transformado em código Kermeta utilizando a funcionalidade disponibilizada pelo *plugin* da linguagem para a plataforma Eclipse.

A figura abaixo mostra um trecho dessa transformação:




```
class Model
{
    attribute Packages : Package[0..*]

}
class Package inherits NamedEntity
{
    attribute Entities : DefinitionEntity[0..*]

}
class Class inherits DefinitionEntity
{
    attribute Methods : Method[0..*]

    attribute Attributes : Attribute[0..*]
}
```

```
class Protocol inherits DefinitionEntity
{
    attribute Incomings : Signal[0..*]

    attribute Outgoings : Signal[0..*]

    attribute StateMachine : StateMachine

}
class BehaviouralCapsule
inherits CapsuleDefinition
{
    attribute StateMachine : StateMachine
    attribute Methods : Method[0..*]
    attribute Attributes : Attribute[0..*]
}
```

Figura 3.17 - Transformação Metamodelo-Kermeta

Na Figura 3.17 parte do metamodelo proposto foi transformada de modo automático em linguagem Kermeta. O mesmo foi feito para o metamodelo por completo.

3.4.3 Arquitetura das transformações

Transformações simples são aquelas que fazem pequenas alterações no modelo, geralmente remoções, inserções ou atualizações de instâncias e não necessitam de outras transformações para serem definidas. As compostas ou complexas são aquelas definidas a partir de um conjunto de transformações simples. Dado isso, foi proposta uma arquitetura para as transformações simples e um modelo estendido desta para as compostas. A figura abaixo mostra o diagrama de classes para a transformação simples compor protocolos:

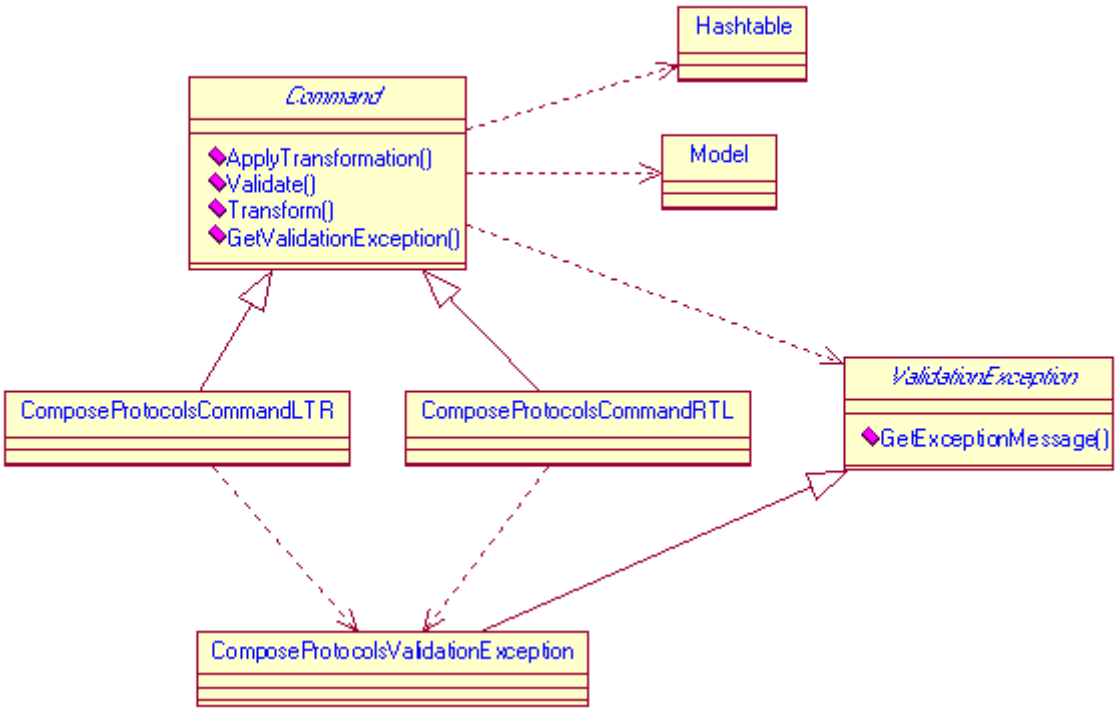


Figura 3.18 - Diagrama de classes para transformação simples

Foi utilizado o padrão de projeto *Command*. Este padrão encapsula o conceito de comando dentro de um objeto. Um objeto que deseje requisitar comandos guarda uma referência para um comando genérico. Quando deseja executar algum comando, apenas instancia o objeto que possui a definição do comando particular. Na Figura 3.18, *Command* representa um comando genérico que possui a assinatura de três operações comuns a todos os tipos de transformações:

```
operation Validate(model : Model, parameterMap : Hashtable<String, Object>) :
Boolean is abstract

operation Transform(model : Model, parameterMap : Hashtable<String, Object>) :
Model is abstract

operation GetValidationException() : ValidationException is abstract
```

Figura 3.19 - Operações comuns a todos os tipos Command

Antes que qualquer transformação seja invocada, é preciso validar o modelo de entrada para verificar se este satisfaz as pré-condições impostas. Caso o modelo seja válido, a transformação segue normalmente, caso contrário uma mensagem de erro deve informar o tipo de erro ocorrido. Na Figura 3.19 a assinatura da operação *Validade()* recebe um modelo (*Model*) de entrada e um conjunto de mapeamentos (*Hashtable<String,Object>*) com os dados necessários para a transformação, verificando sua validade. A operação *Transform()* recebe os mesmos parâmetros que *Validate()* gerando como saída o modelo transformado; vale salientar que esta operação tem efeito colateral e altera o modelo de entrada. A operação *GetValidationException()* retorna um subtipo da exceção genérica *ValidadeException* em caso de não atendimento às pré-condições impostas.

Cada transformação em particular implementa sua própria lógica de validação, transformação e geração do tipo de exceção adequada. Porém todas as transformações sempre fazem a chamada do *Validate()* seguida do *Transform()* em caso de sucesso ou *GetValidationException()* caso contrário. Sendo assim, *Command* define uma operação concreta que representa esse fluxo de execução comum a todas as transformações, como mostrado na figura abaixo:

```
operation ApplyTransformation(model: Model, parameterMap : Hashtable<String, Object >) :
Model is do
    if Validate(model, parameterMap) then
        model := Transform(model, parameterMap)
    else
        raise GetValidationException()
    end
    result := model
end
```

Figura 3.20 - Fluxo de execução comum a todos as transformações

Todas as transformações simples seguem a mesma estrutura apresentada na Figura 3.18. Para as transformações compostas é preciso uma estrutura um pouco mais sofisticada, a estrutura desenvolvida neste trabalho é mostrada na figura abaixo:

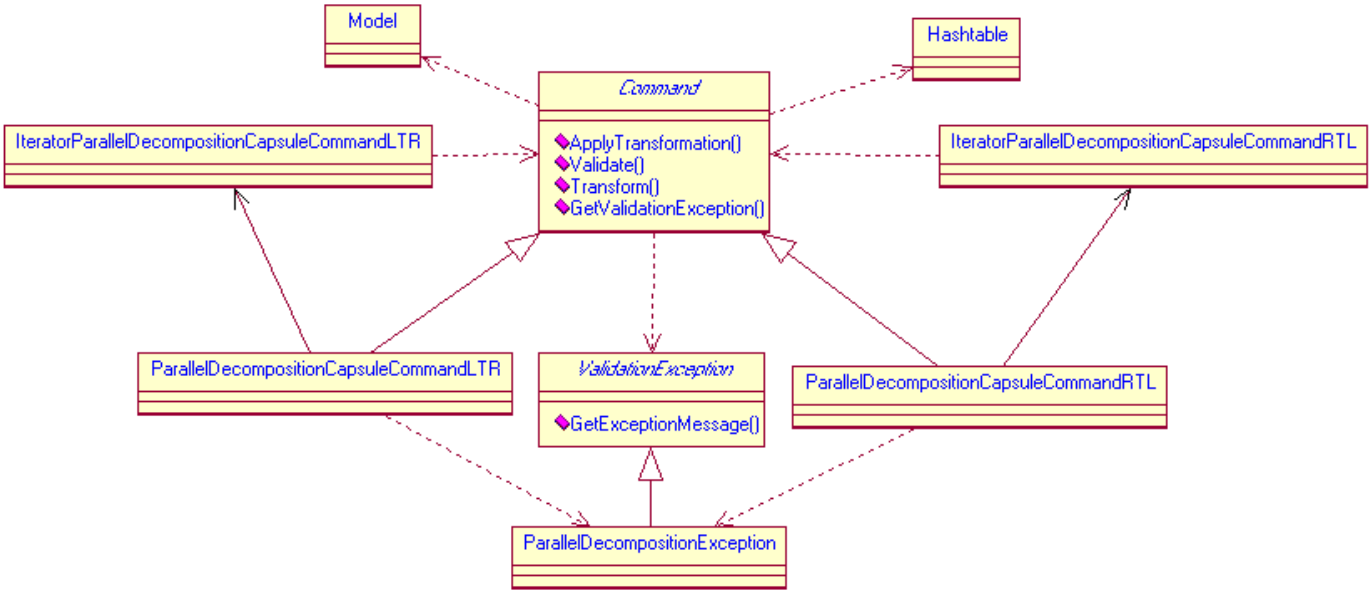


Figura 3.21 - Diagrama de classes transformação composta Decomposição Paralela de uma cápsula

As transformações simples, que compõe uma transformação composta, devem ser aplicadas numa ordem pré-estabelecida sobre os modelos intermediários em cada etapa da transformação composta. A Figura 3.21 mostra o diagrama de classes que captura esse conceito. *ParallelDecompositionCommmandLTR* e *ParallelDecompositionCommmandRTL* representam os comandos que executam as duas direções da transformação. Cada um deles está associado a um *iterator* (*IteratorParallelDecompositionCommmandLTR* e *IteratorParallelDecompositionCommmandRTL*) que possui as operações *hasNext()* e *next()*. A primeira delas verifica a existência de alguma transformação que ainda resta para ser aplicada ao modelo, enquanto a segunda retorna um *Command* representando a próxima transformação simples que deve ser aplicada.

Os comandos que representam transformações paralelas possuem os mesmos métodos que transformações simples: *Validate()*, *Transform()* e *getValidationException()*. O retorno do *Validate()* das transformações compostas, abordadas nesse trabalho, sempre retorna um valor verdadeiro, indicando que a transformação pode prosseguir. Isso faz com que a real validação seja feita pelas transformações simples que formam a composta.

3.4.4 Linguagem de representação dos modelos

A linguagem escolhida para representar os modelos a serem manipulados (lidos, transformados e persistidos) foi à adotada pela OMG para tal fim: XMI (*XML Metadata Interchange*) .

XMI é uma extensão de XML que permite codificação de modelos arbitrários cuja linguagem é definida por metamodelos estendidos de Ecore. Este é exatamente o caso deste trabalho. Além disso, Kermeta permite escrever e ler de modo bastante simples modelos escritos no formato XMI. Por esses motivos escolhemos XMI como formato de representação dos modelos manipulados.

A figura abaixo mostra parte de um modelo UML-RT gerado a partir de um script de criação em Kermeta:

```
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001,
  <Packages name="Package_Compose_Protocols_Model">
    <Entities xsi:type="metamodel:Protocol" name="protocol_X">
      <Incomings name="inc_x"/>
      <Outgoings name="out_x"/>
    </Entities>
    <Entities xsi:type="metamodel:Protocol" name="protocol_Y">
      <Incomings name="inc_y"/>
      <Outgoings name="out_y"/>
    </Entities>
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="capsule_A">
      <Ports xsi:type="metamodel:EndPort" name="end_port_a_1" protocol="//@Packages.0/@Entities.0"/>
      <Ports xsi:type="metamodel:EndPort" name="end_port_a_2" protocol="//@Packages.0/@Entities.1"/>
      <StateMachine>
        <States name="state_a"/>
      </StateMachine>
      <Methods name="met_a"/>
      <Attributes name="att_a"/>
    </Entities>
```

Figura 3.22 - Parte de um modelo UML-RT em XMI

Um script de geração de modelos em Kermeta possui quatro etapas: criação de um repositório para um conjunto de modelos, criação de um recurso no repositório para um modelo particular, criação de objetos que representem os elementos do mesmo e suas relações, seguido de sua persistência.

A Figura 3.23 exibe parte do código responsável pela geração do modelo mostrado na Figura 3.22. O primeiro passo é a criação de um repositório de modelos, *repository*, seguido pela criação de um recurso, *resource*, para o modelo *ComposeProtocolsModel-IN.xmi*, que tem sua semântica definida no metamodelo *Metamodel.ecore* (metamodelo definido neste trabalho).

Todos os elementos e relações definidos no metamodelo, como dito anteriormente, foram mapeados equivalentemente em classes e relações de Kemeta. Ainda na Figura 3.23 alguns desses elementos (*Model*, *Package* e *Protocol*) são instanciados e tem suas propriedades valoradas. Em seguida alguns relacionamentos são estabelecidos: as relações de composição entre *Protocol* e *Package*, e deste com *Model*. Por fim o modelo é persistido.

```
// CRIAÇÃO DO REPOSITÓRIO
 var repository : EMFRepository init EMFRepository.new
 var resource : EMFResource
 //CRIAÇÃO DO MODELO
 resource ?= repository.createResource
 ("ComposeProtocolsModel-IN.xmi","Metamodel.ecore")

 //CRIAÇÃO DOS ELEMENTOS DO MODELO
 var model : Model init Model.new
 var pack : Package init Package.new
 //ATRIBUIÇÃO DE SUAS PROPRIEDADES
 pack.name := "Package_Compose_Protocols_Model"
 var protocol_X : Protocol init Protocol.new
 protocol_X.name := "protocol_X"

 //ESTABELECIMENTO DAS RELAÇÕES
 pack.Entities.add(protocol_X)
 model.Packages.add(pack)
 resource.instances.add(model)

 //PERSISTÊNCIA
 resource.save()
```

Figura 3.23 - Manipulação(criação de modelos em Kermeta)

3.4.5 Visão detalhada da implementação

Na Figura 3.19 é apresentada uma visão caixa preta das operações que compõem todas as transformações simples e compostas. Para um entendimento completo acerca dos detalhes de implementação das transformações, serão apresentados: *template* esquerdo/direito, pré-condições, modelos XMI de entrada e saída e o detalhamento das operações *Validate()*, *Transform()* e *getValidationException()* das transformações: Compor Protocolos(simples) e Extrair Classe (composta). Embora todas as transformações da Tabela 3.1 tenham sido implementadas, apresentamos apenas um lei de cada categoria (simples e composta) como forma exemplificar a estratégia de implementação aplicada para todo o conjunto de leis.

3.4.5.1 Compor Protocolos

Um protocolo estabelece uma interface de comunicação entre duas portas. Durante o refinamento de um modelo UML-RT pode surgir a necessidade de agrupar ou decompor protocolos. Protocolos simples podem ser agrupados em um mesmo protocolo, enquanto um protocolo complexo pode ser dividido em dois ou mais protocolos. A Figura 3.24 exibe a transformação responsável pela composição/divisão de protocolos do ponto de vista declarativo/relacional dos elementos. Na transformação da direita para esquerda, os sinais de entrada (*xincoming*, *yincoming*) e saída(*xoutgoings*,*youtgoings*) dos protocolos *X* e *Y* são agrupados formando o protocolo *XY*. Os protocolos *X* e *Y* deixam de fazer parte do modelo e as portas *p*(protocolo X) e *q*(protocolo *Y*) são substituídas pela porta *pq* com protocolo *XY*. O mesmo ocorre para as portas *r* e *s*. A inversa da transformação subdivide os sinais de entrada e saída do protocolo *XY* nos

protocolos *X* e *Y* criando as portas *p* e *q* em substituição à porta *pq*. O mesmo ocorre para a porta *rs*.

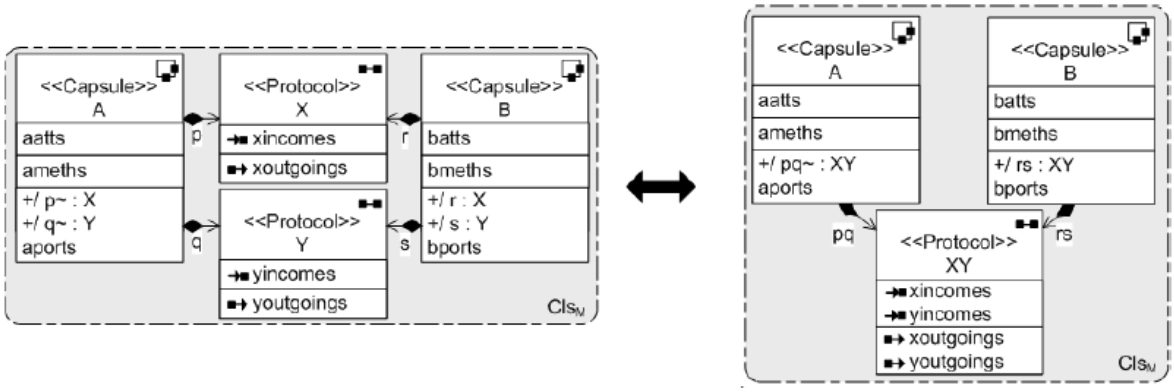


Figura 3.24 -Templates da transformação Compor protocolos, visão declarativa e relacional

A Figura 3.25 mostra o que ocorre do ponto de vista estrutural: para todas as instâncias no modelo Str_M as ligações entre as portas (*p*,*r*) e (*q*,*r*) são substituídas pela ligação entre as portas (*pq*,*rs*), que possuem o novo protocolo XY. A transformação da esquerda para a direita substitui as ligações entre todas as instâncias da cápsula *A* e *B* via portas (*pq*,*rs*) por ligações entre as portas (*p*,*r*) e (*q*,*r*) .

Figura 3.25 - Templates da transformação Compor protocolos, visão estrutural

As pré-condições para aplicação da transformação compor protocolos são apresentadas abaixo:

- (→) *X* e *Y* não possuem sinais com nomes em comum.
- (←) A máquina de estados de *X* utiliza somente os sinais em *xincomes* e *xoutgoings*; A máquina de estados de *Y* utiliza somente os sinais em *yincomes* e *youtgoings*.

Os símbolos → e ← indicam a pré-condição quando a transformação é aplicada da esquerda para a direita e da direita para esquerda respectivamente, e cada um deles está associado a uma operação *Validate()*.

```
method Validate(model : Model, parameterMap : Hashtable<String, Object>) :
Boolean is do
    //PROTOCOLOS
    var prot_1 : Protocol init parameterMap.getValue("protocol_1").asType(Protocol)
    var prot_2 : Protocol init parameterMap.getValue("protocol_2").asType(Protocol)
    var name_collision : Boolean init false
    //VERIFICA ENTRADAS
    prot_1.Incomings.each{s|prot_2.Incomings.each{t|
            if(s.name.equals(t.name)) then
                name_collision := true
            end
        }
    }
    //VERIFICA SAÍDAS
    prot_1.Outgoings.each{s|prot_1.Outgoings.each{t|
            if(s.name.equals(t.name)) then
                name_collision := true
            end
        }
    }
    //RETORNA RESULTADO
    result := (not name_collision)
end
```

Figura 3.27 - Validação da pré- condição → para a transformação Compor Protocolos

```
Model is do
    //...

    //CRIAÇÃO DO PROTOCOLO XY
    var protocol_XY : Protocol init Protocol.new
    protocol_XY.name := "protocol_XY"
    //ATUALIZAÇÃO DAS ENTRADAS
    protocol_XY.Incomings.addAll(protocol_1.Incomings)
    protocol_XY.Incomings.addAll(protocol_2.Incomings)
    //ATUALIZAÇÃO DAS SAÍDAS
    protocol_XY.Outgoings.addAll(protocol_1.Outgoings)
    protocol_XY.Outgoings.addAll(protocol_2.Outgoings)

    //...

    //ATUALIZAÇÃO DAS ENTIDADES NO PACOTE
    pack.Entities.add(protocol_XY)
    pack.Entities.remove(protocol_1)
    pack.Entities.remove(protocol_2)
    result := model
end
```

Figura 3.26 - Operação Transform() da transformação Compor Protocolos

O *Validate()*, mostrado na Figura 3.27, recebe um mapeamento com dois protocolos e verifica se suas entradas e saídas possuem algum sinal em comum.O método retorna valor *true* se a transformação pode prosseguir e *false* caso contrário. Se a transformação puder seguir a operação *Transform()* é chamada sobre o modelo de entrada.

A Figura 3.26 exibe parcialmente a operação *Transform()*, da transformação Compor Protocolos, quando aplicada da esquerda para a direita. No trecho apresentado um novo protocolo é criado, *protocol_XY*, seus sinais são dados pela composição dos sinais dos protocolos *protocol_1* e *protocol_2*, respectivamente *X* e *Y* na Figura 3.28. Em seguida *protocol_1* e *protocol_2* são removidos do pacote dando lugar a *protocol_XY*.

Os modelos (no formato XMI) da figura abaixo exemplificam o uso da transformação compor protocolos:

```
<Packages name="Package_Compose_Protocols_Model">
  <Entities xsi:type="metamodel:Protocol" name="protocol_X">
    <Incomings name="inc_x"/>
    <Outgoings name="out_x"/>
  </Entities>
  <Entities xsi:type="metamodel:Protocol" name="protocol_Y">
    <Incomings name="inc_y"/>
    <Outgoings name="out_y"/>
  </Entities>
```

↓

```
<Packages name="Package_Compose_Protocols_Model">
<Entities xsi:type="metamodel:Protocol" name="protocol_XY">
    <Incomings name="inc_x"/>
    <Incomings name="inc_y"/>
    <Outgoings name="out_x"/>
    <Outgoings name="out_y"/>
  </Entities>
```

Figura 3.28 - Modelo de entrada (acima de ↓) e saída da transformação Compor Protocolos

3.4.5.2 Decomposição Paralela da uma Cápsula

A uma cápsula pode-se atribuir um número crescente de responsabilidades durante o desenvolvimento, o que pode quebrar o seu conceito original ou comprometer seu entendimento. Esses pontos levam à necessidade de uma transformação que bipartcione uma cápsula, dividindo seus métodos, atributos, portas e máquina de estado entre duas novas cápsulas. A decomposição paralela de uma cápsula atua conforme o *template* da Figura 3.29.

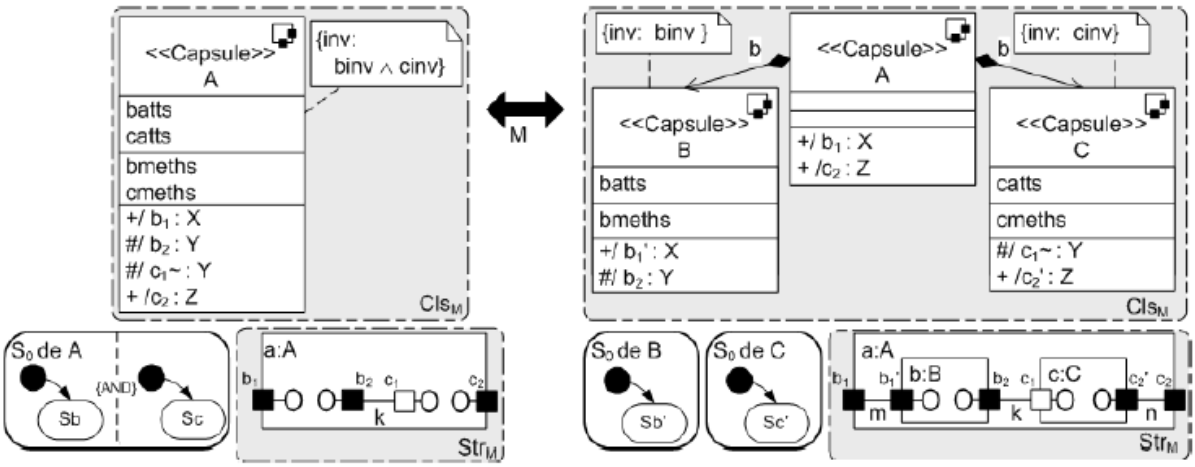


Figura 3.29 - *Template* da transformação Decomposição Paralela de uma cápsula

A transformação da esquerda para a direita atua dividindo atributos, métodos, portas e máquina de estados da cápsula A em dois conjuntos: (*batts, bmeths,* (*b1,b2), Sb*) e (*catts, cmeths,* (*c1,c2), Sc*) que constituirão respectivamente as cápsulas *B* e *C*. Nesse contexto *A* torna-se apenas um intermediador entre o meio externo e as cápsulas que *B* e *C* que juntas oferecem o comportamento inicial provido por *A*. A transformação inversa elimina as cápsulas internas à *A, B* e *C,* fazendo com que esta absorva por completo o comportamento das duas cápsulas.

As pré-condições para aplicação da transformação, Decomposição Paralela de uma Cápsula, são apresentadas abaixo:

- (→) (*batts, binv, bmeths,*(*b1, b2*),*Sb'*) e (*catts, cinv, cmeths,*(*c1, c2*),*Sc'*) particionam *A*.
- (←) As máquinas de estado *Sb* e *Sc* são,respectivamente, isomórficas à *Sb'* e *Sc'*

A partição de uma cápsula *P* implica na existência *n* (*n* >= 2) cápsulas P_i(*atts, inv, meths, ports, stateMachine*) e uma cápsula *P´*, onde , cada cápsula P_i (i Є ℕ|2 <= i <= n]) possui parte do comportamento e estrutura de *P* e *P´* provê, através de cada P_i, o mesmo comportamento de *P*.

A operação *Validate()* da transformação no sentido (→) deixa a cargo das transformações simples a verificação das pré-condições, retornando o mesmo valor de verdade para os modelos de entrada. A Figura 3.30 apresenta o método *Transform()* para a Decomposição paralela de uma Cápsula. A variável *iterator* do tipo *IteratorParallelDecompositionCommandLTR* representa um iterador sobre as transformações simples que compõe a decomposição paralela. Dados os parâmetros da transformação, o *iterador* é responsável pela construção do *Command* apropriado a cada passo da transformação, com base nos parâmetros recebidos pela operação *Transform()*. As operações *hasNextTransformation()* e *nextTransformation()* retornam, respectivamente, um valor de verdade sobre a existência de mais transformações simples a serem aplicadas e os parâmetros da próxima transformação caso ela exista.

```
method Transform(model : Model, parameterMap : Hashtable<String, Object>) : Model is do
    from var iterartor : IteratorParallelDecompositionCapsuleCommandLTR init
       IteratorParallelDecompositionCapsuleCommandLTR.new.initialize(parameterMap)
       //CONDIÇÃO DE PARADA
     until not iterartor.hasNextTransformation
     loop
        var param : Hashtable<String, Object>
        //PROXIMA TRANSFORMAÇÃO:COMMAND E PARAMETROS
        param :=  iterartor.nextTransformation()
        if iterartor.hasNextTransformation then
            var com : Command init param.getValue("command").asType(Command)
            //APLICAÇÃO DA TRANSFORMAÇÃO SIMPLES
            model := com.ApplyTransformation(model,param)
        end
     end
 result := model
end
```

Figura 3.30 - Operação Transform() da transformação composta Decomposição Paralela de uma Cápsula

```
<Entities xsi:type="metamodel:BehaviouralCapsule" name="A">
  <Ports xsi:type="metamodel:EndPort" name="endport_1" protocol="//@Packages.0/@Entities.2"/>
  <Ports xsi:type="metamodel:EndPort" name="endport_2" protocol="//@Packages.0/@Entities.4"/>
  <Ports xsi:type="metamodel:RelayPort" name="relayport_1" protocol="//@Packages.0/@Entities.3"/>
  <Ports xsi:type="metamodel:RelayPort" name="relayport_2" protocol="//@Packages.0/@Entities.3"/>
     ...          ...          ...
  <Methods name="met_1_b" type="//@Packages.0/@Entities.5">
    <parameters type="//@Packages.0/@Entities.5"/>
  </Methods>
  <Methods name="met_2_b" type="//@Packages.0/@Entities.5"/>
  <Methods name="met_1_c" type="//@Packages.0/@Entities.5"/>
  <Methods name="met_2_c" type="//@Packages.0/@Entities.5"/>
  <Attributes name="att_1_b" type="//@Packages.0/@Entities.5"/>
  <Attributes name="att_2_b" type="//@Packages.0/@Entities.5"/>
  <Attributes name="att_3_b" type="//@Packages.0/@Entities.5"/>
  <Attributes name="att_1_c" type="//@Packages.0/@Entities.5"/>
  <Attributes name="att_2_c" type="//@Packages.0/@Entities.5"/>
</Entities>
```

Figura 3.32 - Parte do modelo de entrada da transformação Decomposição Paralela de uma Cápsula

```
<Entities xsi:type="metamodel:BehaviouralCapsule" name="B">
  <Ports xsi:type="metamodel:EndPort" name="endport_1" protocol="//@Packages.0/@Entities.1"/>
  <Ports xsi:type="metamodel:EndPort" name="relayport_1" protocol="//@Packages.0/@Entities.2"/>
  <StateMachine xsi:type="metamodel:StateMachine">
     ...          ...
  <Methods name="met_1_b" type="//@Packages.0/@Entities.4">
    <parameters type="//@Packages.0/@Entities.4"/>
  </Methods>
  <Methods name="met_2_b" type="//@Packages.0/@Entities.4"/>
  <Attributes name="att_1_b" type="//@Packages.0/@Entities.4"/>
  <Attributes name="att_2_b" type="//@Packages.0/@Entities.4"/>
  <Attributes name="att_3_b" type="//@Packages.0/@Entities.4"/>
</Entities>
<Entities xsi:type="metamodel:BehaviouralCapsule" name="C">
  <Ports xsi:type="metamodel:EndPort" name="endport_2" protocol="//@Packages.0/@Entities.3"/>
  <Ports xsi:type="metamodel:EndPort" name="relayport_2" protocol="//@Packages.0/@Entities.2"/>
     ...          ...
  <Methods xsi:type="metamodel:Method" name="met_1_c" type="//@Packages.0/@Entities.4"/>
  <Methods xsi:type="metamodel:Method" name="met_2_c" type="//@Packages.0/@Entities.4"/>
  <Attributes xsi:type="metamodel:Attribute" name="att_1_c" type="//@Packages.0/@Entities.4"/>
  <Attributes xsi:type="metamodel:Attribute" name="att_2_c" type="//@Packages.0/@Entities.4"/>
</Entities>
<Entities xsi:type="metamodel:BehaviouralCapsule" name="A">
  //referencias para as capsulas B,e C
  //portas de conexão
</Entities>
```

Figura 3.31 -Parte do modelo de entrada da transformação Decomposição Paralela de uma Cápsula

Os artefatos da Figura 3.31 e Figura 3.31 apresentam parte dos modelos de entrada e saída da transformação. A *BehaviouralCapsule A* tem seus métodos, atributos, portas e máquina de estados(omitida) divididos entre as cápsulas *B* e *C* que em seguida tornam-se atributos de *A*. Isso refletirá no diagrama estrutural na medida em que toda instância de *A* terá como cápsulas internas *B* e *C*.

O detalhamento das transformações apresentadas na Tabela 3.1- Leis implementadas, a lista de transformações simples e a sua ordem de aplicação para cada lei composta estão presentes em [45].

3.5 Contribuições e Problemas Encontrados

Um estudo de caso referente à aplicação das leis de transformação simples e composta foi desenvolvido em [45] e implementado como forma de validação das leis automatizadas neste trabalho.

A contribuição deste trabalho no que se refere a transformações de modelos foi o desenvolvimento de uma estratégia de implementação de transformações em modelos UML-RT e da composição destas, em transformações mais complexas, de forma suficientemente genérica e reutilizável para desenvolvimento de transformações em outras linguagens de modelagem.

Um dos problemas encontrados é que a busca pelos pontos do modelo onde se iria aplicar a transformação não foi automatizada em trabalhos anteriores. Surge assim, a necessidade de desenvolvimento de uma *engine* de busca por padrões arquiteturais (*templates* de casamento) onde é possível aplicar uma dada transformação. Como na aplicação de *refactorings*, geralmente se está interessado em encontrar todos os pontos no modelo que sigam uma data estrutura, um requisito essencial é que a *engine* retorne todos os casamentos possíveis. Na próxima seção é apresentada uma implementação desta *engine* suficientemente abrangente para buscar qualquer padrão em qualquer modelo, não se limitando a UML-RT. Essa é sem dúvida a maior contribuição deste trabalho.

4 Casamento de Padrões

Muitos problemas em computação estão associados de alguma forma ao casamento de padrões. Em um compilador, por exemplo, a análise léxica procura por padrões em uma seqüência de caracteres para gerar uma seqüência de *tokens* [23]. Neste caso a gramática léxica define um conjunto de padrões que tentaram ser casados em uma arbitrária seqüência de caracteres.

Formalmente, um padrão é uma regra de formação que pode estar ou não sendo seguida pelos elementos que constituem o espaço de busca. Na Figura 4.1 é mostrado um exemplo simples: O padrão apresentado tem definidas as abstrações *dig* e *num* e suas regras de formação. As duas instâncias formadas apenas por dígitos numéricos no *Espaço de Busca* podem ser geradas a partir das regras de formação e, portanto, casam com padrão definido pela abstração *num*.

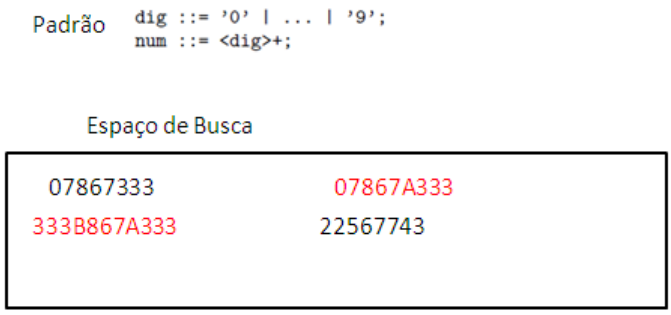


Figura 4.1 - Padrão e espaço de busca

As regras de formação apresentadas no padrão da figura acima são definidas a partir dos próprios elementos que constituem as entradas do espaço de busca, porém nem todos os elementos do espaço de busca estão necessariamente presentes nas regras de formação.

4.1 Padrões arquiteturais

Dado um metamodelo e um conjunto de modelos, um padrão arquitetural define uma abstração cuja regra de formação é expressa em função dos elementos e relações possíveis nos modelos que constituem o espaço de busca.

A Figura 3.2 exibe a definição de um padrão arquitetural simples que casa com todas as instâncias de *ANode* presentes no modelo da Figura 3.3 (espaço de busca).

4.1.1 Definição de padrões arquiteturais

Um padrão arquitetural é, antes de tudo, um modelo construído conforme um metamodelo *M'*. Um metamodelo *M*, que defina os modelos do espaço de busca, é a base de construção de *M´* (Figura 4.2), que se dá removendo as restrições de formação e relacionamento dos elementos de *M*.

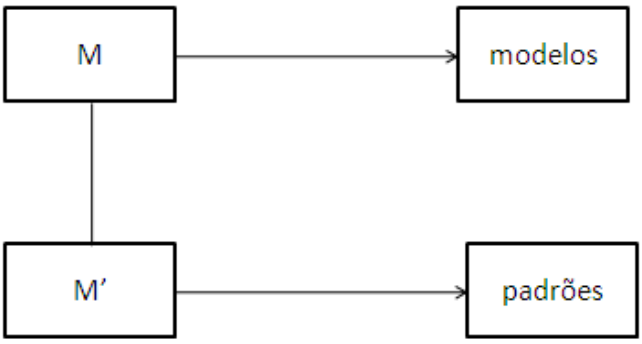


Figura 4.2 - Relação entre metamodelos M e M'

Na definição formal de UML-RT toda cápsula tem obrigatoriamente um atributo *name* em um modelo válido. Um padrão para buscar todas as cápsulas num dado espaço de busca exige que a restrição sobre o atributo *name* seja eliminada no

metamodelo (*M´*) que define os padrões, assim uma cápsula em *M´* pode ou não possuir o atributo *name*.

De modo geral um metamodelo M´ é derivado a partir de M eliminando-se toda e qualquer obrigatoriedade em relação às propriedades (por exemplo, *name*) e relações(por exemplo, toda classe deve está associada a um pacote) . Assim um padrão (modelo), com uma única instância de *Class* (sem a propriedade *name*), seria válido conforme M' e casaria com todas as classes de todos os pacotes nos modelos do espaço de busca (Figura 4.3 a). Outro padrão formado com uma instância de *Package (name='X')*, possuindo uma instância de *Capsule* (sem a propriedade *name*) através da relação *Entities*, casa com todas as cápsulas que estão no pacote *X*(Figura 4.3 b).

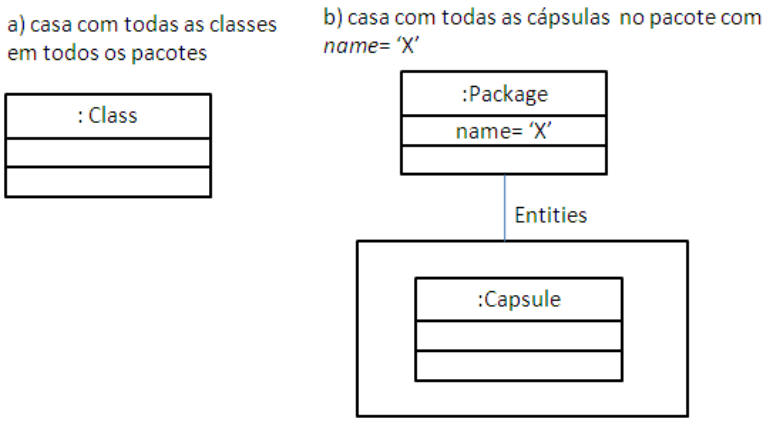


Figura 4.3 - Padrões para modelos UML-RT

Padrões podem ser mais ou menos restritivos até o ponto em que se tornem modelos *M* válidos. De fato modelos *M* podem ser considerados padrões já que são apenas modelos M´ onde foram impostas todas as restrições optativas(obrigatórias em M).

Embora os exemplos de padrões apresentados até aqui sejam referentes à linguagem UML-RT a API de casamento de padrões arquiteturais desenvolvida neste trabalho é suficientemente geral e pode ser aplicada a qualquer tipo de metamodelo definido como uma extensão de Ecore.

4.2 API para casamento de padrões

O casamento de padrões desenvolvido nesse trabalho é de propósito geral e suas funcionalidades/serviços podem ser utilizadas por programas aplicativos de modo transparente.

De modo geral, a API funciona recebendo como entrada, um metamodelo arbitrário *M*, um modelo válido segundo *M* e um padrão *P* e gera como saída uma lista de todos os pontos de casamento no modelo com dado *P*(Figura 4.4).

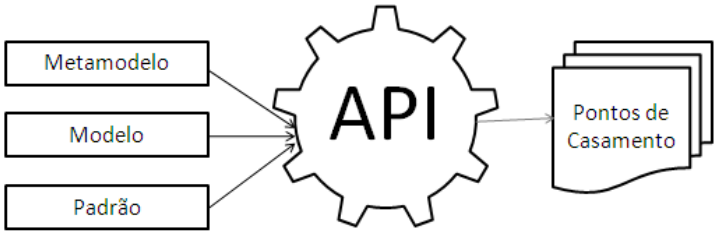


Figura 4.4 - Visão caixa preta da API para casamento de padrões arquiteturais

A API foi desenvolvida em três etapas: transformação do metamodelo e modelo para uma linguagem de representação do conhecimento LRC, transformação do padrão na linguagem de consulta para a LRC e por fim tratamento dos resultados do casamento.

Para a escolha da LRC foram considerados três critérios:

- Poder de representação equivalente aos metamodelos estendidos de Ecore,
- Facilidade de desenvolvimento da conversão de metamodelos e modelos para a LRC, assim como transformação do padrão de casamento para a linguagem de consulta da mesma.
- Facilidade de integração com o framework Kermeta, dado que foi o escolhido para implementação das transformações.

4.3 FLORA-2

Flora-2 é uma linguagem orientada a objetos para construção e consultas em bases de conhecimento, além de uma plataforma para o desenvolvimento de aplicações nas áreas de agentes inteligentes, Web Semântica, gerenciamento de ontologias entre outros [44].

A linguagem é implementada através de um compilador que traduz o código Flora-2 para Prolog. Flora-2 é um dialeto de F-Logic (*Frame Logic*) [59] com extensões que inclui meta-programação semelhante a HiLog [59] e atualizações lógicas semelhantes a Lógica Transacional[26,27] .

```
male :: person.     // relação de sub classe
female :: person.   // relação de sub classe

person  [              //propriedades
    mother{1:1}*=>person, father{1:1}*=>person,
    spouse{1:1}*=>person, husband{1:1}*=>person,
    son*=>person, daughter*=>person,
    brother*=>person, sister*=>person,
    uncle*=>person, aunt*=>person,
    ancestor*=>person, parent*=>person
        ].
```

Figura 4.5 - Metamodelo em Flora

A Figura 4.5 mostra um metamodelo *MF* em flora. Este possui relações de subtipo (*male :: person* indica que *male* é subtipo de *person*), definição de propriedades com cardinalidade unária (*mother{1:1}*=>person* indica que uma instância de *person* está associada a apenas uma instância de *mother*) e relação com cardinalidade n-ária(*son*=>person* indica que uma instância de *person* esta associada a zero ou mais instâncias de *son*).

```
rita:female.
franz:male[mother->rita,father->wilhelm].
heinz:male[mother->rita,father->wilhelm].
hermann:male[mother->rita,father->wilhelm,spouse->johanna].
johanna:female.
monique:female.
bernhard:male[mother->johanna,father->hermann].
karl:male[mother->johanna,father->hermann,spouse->christina].
christina:female.
kati:female[mother->johanna,father->hermann].
albert:male[mother->monique,father->bernhard].
eva:female[mother->kati].
```

Figura 4.6 - Modelo válido conforme MF

A Figura 4.6 apresenta um modelo válido conforme *MF*. A expressão *rita:female* cria uma instância *rita* de *female* (subtipo de *person*). A declaração *franz:male[mother->rita,father->wilhelm]* cria uma instância de *male*, chamada *franz*, e associa ao atributo *mother* a instância *rita*, e ao atributo *father* o valor *wilhelm*(*wilhelm* representa aqui apenas um valor *string*, diferente de *rita* que é de fato uma instância *person*).

É possível estabelecer relações de equivalência e com isso valorar outras relações. Na expressão *?X[spouse->?Y] :- ?Y[spouse->?X]* é definida a simetria na relação *spouse*: sempre que *?Y* for *spouse* de *?X* então *?X* será *spouse* de *?Y*. Esta relação é capaz de preenche atributos em algumas instâncias derivando essa informação da base de conhecimento através da definição dessas regras.

Por fim, é possível efetuar uma série de consultas como *franz[mother->?X,father->?Y]* que retornará os pais de *frank* : *?X = rita* e ?= *wilhelm*.

Embora apresente um completo ambiente para definição de metamodelo, modelos e consultas Flora-2 apresenta sérias complicações de utilização:

- Necessidade da instalação previa de XSB [60], um sistema para programação lógica limitado ao Unix e Windows, com uma instalação bastante complicada
- Não existência de uma API de comunicação entre Flora e Kermeta, que possibilitasse o uso das transformações desenvolvidas na primeira parte deste trabalho
- Dificuldade tanto de conversão de modelos e metamodelos para Flora-2 como de transformação de padrões em *querys*, dada a disparidade da linguagem em relação às linguagens de definição dos modelos, metamodelos e padrões.

Dados os problemas apresentados por FLORA-2, outras linguagens de representação de conhecimento foram estudantes e, dentre elas, a que melhor atendeu aos critérios supracitados foi OWL-SPARQL, como explicado nas próximas secções.

4.4 OWL, SparQL

A W3C(*World Wide Web Consortium*) [62] é um consórcio de empresas de tecnologia que tem como objetivo desenvolver especificações, *guidelines*, softwares e ferramentas que permitam a evolução da Web e assegurem sua interoperabilidade. Com o advento da web semântica a W3C criou a linguagem OWL (*Web Ontology Language*) [61] que permite a representação explicita do significado da informação, facilitando o processamento e integração de modo automático das informações na Web.

OWL oferece suporte à definição de classes e relações entre elas (que estão intrínsecas em documentos e aplicações Web). Com OWL é possível definir domínios através da criação de classes e suas propriedades, definição de instâncias de classes e criação asserções sobre elas.

OWL faz parte de uma pilha da pilha de especificações da W3C como mostrado na Figura 4.7:

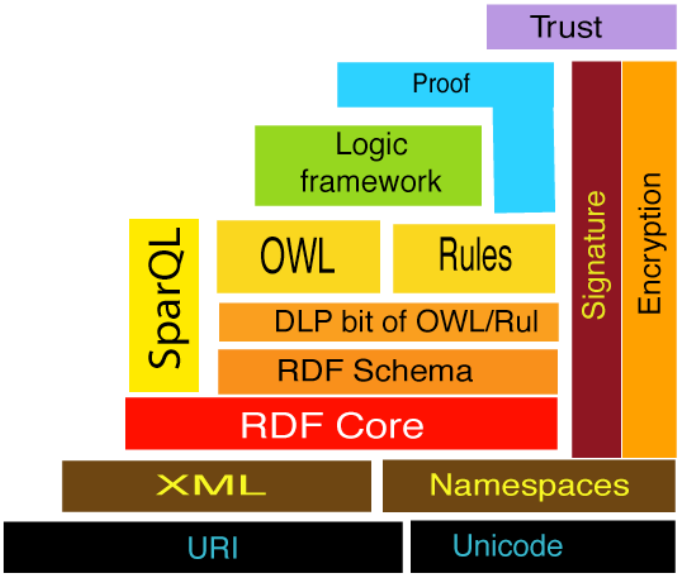


Figura 4.7 - Pilha de especificações W3C

Na figura acima, OWL se apresenta como uma linguagem estruturada sobre XML e RDF, estendendo estas com objetivo de obter maior poder de abstração e expressividade:

XML provê uma sintaxe para documentos estruturados, mas é impossível definir restrições semânticas sobre o significado desses documentos. XML Schema é uma linguagem para restringir a estrutura de documentos XML estendendo XML com tipos de dados.

RDF é um modelo de dados para objetos e as relações entre eles, provendo uma semântica simples para modelos de dados, sendo estes representados em uma sintaxe XML. RDF Schema é um vocabulário para descrever propriedades e classes de objetos RDF com uma semântica para hierarquia de generalização de classes e propriedades.

Neste contexto, OWL adiciona mais vocabulário a RDF para descrever propriedades e classes, como: relação de disjunção entre classes, restrições de cardinalidade (*minCardinality*, *maxCardinality*), (não) igualdade (*sameAs, differentFrom ,AllDifferent*), características das propriedades(*ObjectProperty, DatatypeProperty*, *inverseOf, TransitiveProperty, SymmetricProperty*) e tipos enumerados.

A Figura 4.8 exibe um metamodelo simples de uma maquina de estados *FSM* com as propriedades *hasStates* (cunjunto de *States*) e *hasTransitions* (conjunto de *Transsition*).

```
//definição de classes
<owl:Class rdf:ID="Transition"/>
<owl:Class rdf:ID="State"/>
<owl:Class rdf:ID="FSM"/>
//definição de propriedades
<owl:ObjectProperty rdf:ID="hasStates">
  <rdfs:domain rdf:resource="#FSM"/>
  <rdfs:range rdf:resource="#State"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasTransitions">
  <rdfs:domain rdf:resource="#FSM"/>
  <rdfs:range rdf:resource="#Transition"/>
</owl:ObjectProperty>
```

Figura 4.8 - Metamodelo de uma FSM OWL

A especificação de RDF define a sintaxe e semântica de uma linguagem de *query* (SparQL [53]) para informações no formato RDF. SparQL pode ser usada para expressar consultas sobre diversas fontes de dados armazenados em formato RDF ou OWL ou vistos assim através de um *middleware*.

A consulta SparQL abaixo define as variáveis para casamento *?X, ?Y,?Z* e uma série de restrições de tipo: Dado a clausula *?X :hasTransitions ?Y* então *?X* é necessariamente do tipo *FSM* e *?Y* do tipo *Transition* o mesmo acontece em que *?X :hasStates ?Z* onde *?Z* é necessariamente do tipo *State*. Essa consulta retornara todas as instâncias de *FSM* e para cada uma delas o conjunto de todos os seus estados e transições.

SELECT ?X ?Y ?Z WHERE {
?X :hasTransitions ?Y.
?X :hasStates ?Z}

Tanto OWL como Ecore tem XML como uma origem em comum o que torna natural a conversão entre Ecore e OWL. Além disso, existe um amplo suporte de para a definição e manipulação de metamodelos e modelos OWL em API´s escritas em Java e já que Kermeta possui uma comunicação simples e direta com Java isso possibilita, por transitividade, uma comunicação entre Kermeta e essas API´s. Por

esses motivos, OWL foi escolhida como linguagem a ser utilizada como suporte para implementação do casamento de padrões arquiteturais junto com SparQL.

4.5 Arquitetura da API de casamento de padrões

Para implementação do casamento de padrão foram criados dois projetos complementares: *patternmatching.backend* (Java) e *patternmatching.frontend* (Kermeta). Em linhas gerais, o primeiro provê uma infra-estrutura para conversão de metamodelos Ecore e modelos XMI para OWL bem como a transformação de padrões em consultas SparQL. O segundo projeto é responsável por utilizar coordenadamente os serviços do primeiro e assim prover, de modo transparente, os serviços de casamento de padrão arquitetural para programas aplicativos.

A estrutura de classes do projeto *patternmatching.backend* é apresentada na Figura 4.9.

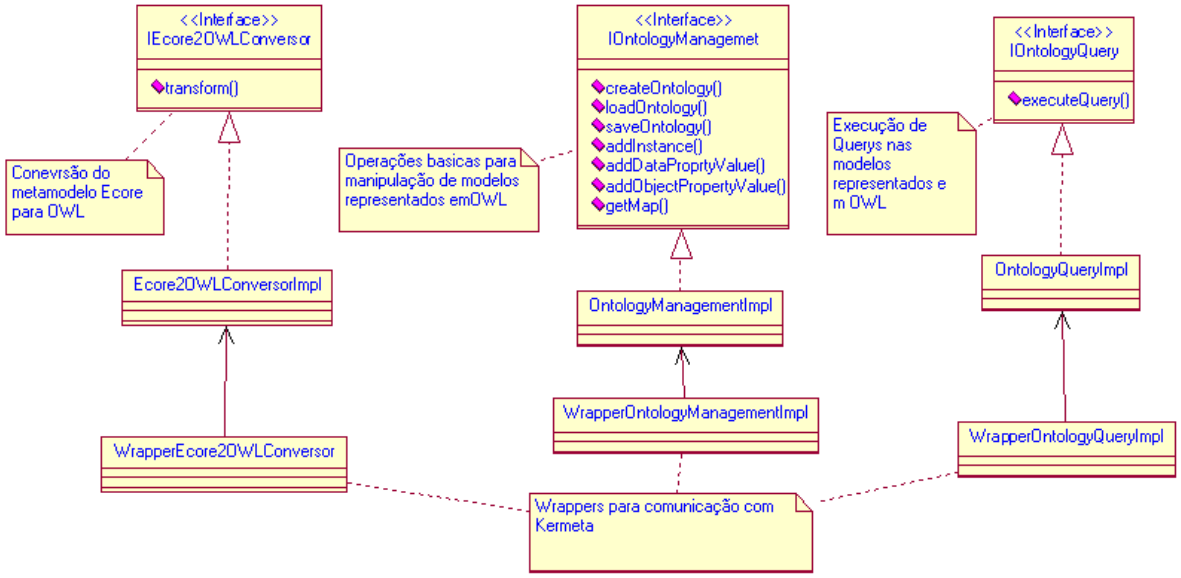


Figura 4.9 - Digrama de classes para o projeto patternmatchin.backend

O serviço de conversão de metamodelos Ecore para OWL é provido pelas classes *IEcore2OWLConversor* e *Ecore2OWLConversorImpl*. A interface *IEcore2OWLConversor* define o assinatura do método *transform()* que é implementado na classe *Ecore2OWLConversorImpl* utilizando a API EODM[54].

A infra-estrutura para conversão de modelos XMI em OWL é provida pelo conjunto de classes *IOntologyManagemet* e *OntologyManagementImpl*. A interface *IOntologyManagemet* possui as assinaturas dos métodos responsáveis pela criação, leitura e persistência de ontologias (metamodelo Ecore em formato OWL), bem como criação de instâncias e valoração de suas propriedades. Por fim, as classe *IOntologyQuery* e *OntologyQueryImpl* são responsáveis pela execução de *queries*(representando padrões).

As classes *WrapperEcore2OWLConversor*, *WrapperOntologyManagementImpl* e *WrapperOntologyQueryImpl* são os pontos de comunicação entre *patternmatching.backend* (Java) e *patternmatching.frontend* (Kermeta). Como Kermeta é inteiramente escrita em Java, existe uma interface bem definida de comunicação entre as duas: *RuntimeObject*. *RuntimeObject* permite que código em Kermeta invoque métodos Java passando objetos Kermeta como parâmetro, que são então convertidos em objetos nativos Java. A interface também permite que objetos Java, retornados na chamada de métodos, sejam convertidos em objetos Kermeta. A Figura 4.10 exemplifica o uso da interface *RuntimeObjetc* em *WrapperOntologyImpl.createOntology(RuntimeObject)*, que dado o parâmetro *fileInput*, vindo de uma chamada ao método em Kermeta, converte o objeto em um String nativa de Java para chamada do método *IOntologyManagement.createOntology(String)*. O mesmo ocorre com os demais *wrappers*.

```
public interface IOntologyManagemet {

    public void createOntology(String fileInput)throws FileNotFoundException;

-------------------------------------------------------------------------------

public class WrapperOntologyManagementImpl {
  private static IOntologyManagemet management;

  public static void createOntology(RuntimeObject fileInput)
  throws FileNotFoundException {
        management.createOntology((String)fileInput.getJavaNativeObject());

  }
```

Figura 4.10 - *Wrapper* de comunicação com Kermeta

A arquitetura do projeto *patternmatching.frontend*(Kermeta) é apresentada abaixo:

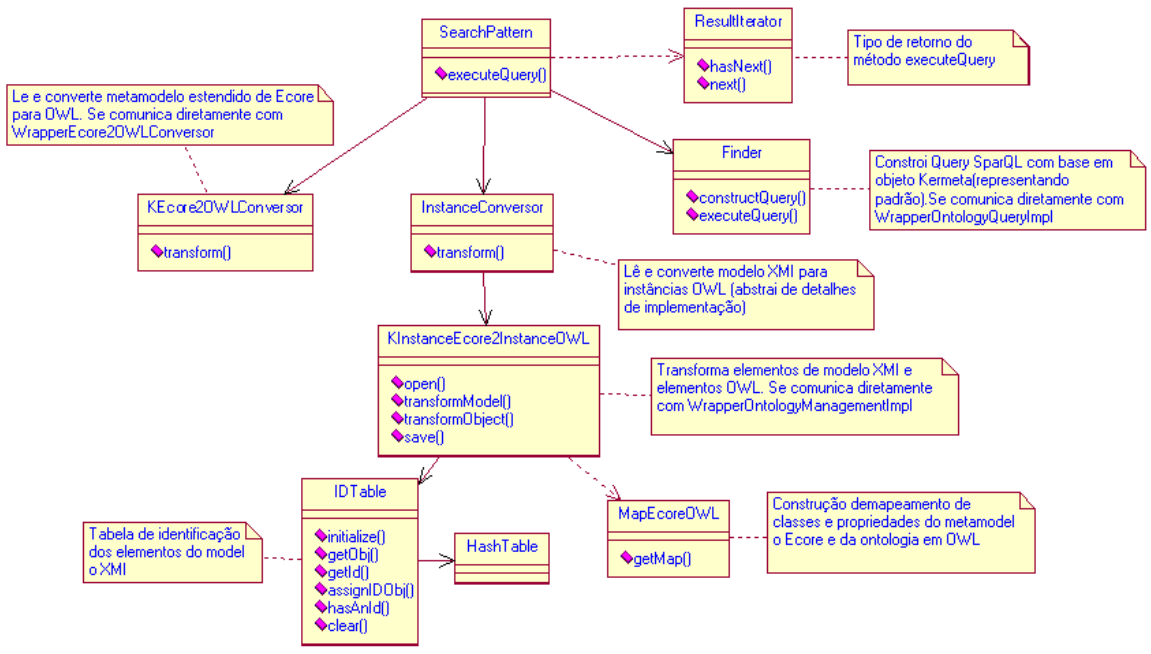


Figura 4.11 - Diagrama de classes para o projeto patternmatching.frontend

Na Figura 4.11 *KEcore2OWLConversor* é responsável por fazer a leitura do metamodelo e invocar os métodos providos por *WrapperEcore2OWLConversor* para conversão do metamodelo. A Figura 4.12 exemplifica uma chamada de método Java em Kermeta. A palavra reservada *extern* indica que a sentença seguinte é uma chamada a um método Java. Ainda na Figura 4.12 o método *conversor()* é chamado com objetos Kermeta, que serão convertidos em objetos nativos Java na execução do método.

```
operation transform( ecorePath : String, owlPath : String ): Void is do
    extern wraper::conversor::WrapperEcore2OWLConversor.transform(ecorePath,owlPath)
end
```

Figura 4.12 - Chamada de método Java em Kermeta

InstanceConversor é responsável pela leitura e conversão dos modelos em instâncias OWL. Para tal utiliza os serviços oferecidos por *KInstanceEcore2InstanceOWL*, que dado um conjunto de objetos que representem elementos de um modelo, converte-os em instâncias e propriedades OWL através do uso de *WrapperOntologyManagementImpl*.

```
operation transformModel(objects: Set<Object>) : Void is do
   objTable.clear
   objects.each { o | transformObject(o)}

end

operation transformObject(obj : Object) : Void is do
       ...

     extern wrapper::ontology::management::WrapperOntologyManagementImpl.addInstance(...)

    // para todas as propriedades da instância
      //se subObjeto ainda não armazenado
       transformObject(obj.subObject)

       // se objectProperty
      extern wrapper::ontology::management::WrapperOntologyManagementImpl.setObjectPropertyValeu(...)

       //se dataProperty
      extern wrapper::ontology::management::WrapperOntologyManagementImpl.addDataProprtyValue(...)
       ...

  end
```

Figura 4.13 - Transformação de elementos de um modelo(conjunto de objetos Kermeta) em OWL

Um modelo XMI quando é lido é representado sobre a forma de um conjunto de objetos Kermeta. Ao ler um modelo XMI, *InstanceConversor* invoca a operação *transformModel()*, passando o conjunto de objetos que representam o modelo lido. Por sua vez, *transformObject()*, converte cada elemento e suas propriedades em elementos e propriedades OWL, através de chamadas a métodos do *wrapper*. Caso um objeto tenha como atributo algum objeto (*objectProperty*), ainda não persistido sob o formato OWL, a função é chamada recursivamente até que o objeto tenha somente propriedades do tipo *dataProperty*, quando a função volta da recursão.

A tabela de identificação, *IDTable*, mantém o registro de quais objetos foram convertidos em OWL para evitar que se tente inserir instâncias duplicadas que são referenciadas por mais de um elemento.

Em OWL, propriedades são estão no escopo global e não podem ser duplicadas, assim surgem alguns problemas na conversão do metamodelo em formato Ecore que admite que existam mais de uma propriedade com o mesmo identificador desde que elas estejam associadas a diferentes classes. Um exemplo disso é a propriedade *Methods*, que representa a relação com um conjunto de métodos tanto em *Class* como em *Capsule* no metamodelo UML-RT proposto.

Para evitar que metamodelos tivessem também essa restrição são criadas duas propriedades diferentes em OWL: *Methods* e *Methods_1*. Assim surgiu a necessidade de se conhecer esse fato durante a conversão de modelos. Para tal foi criada *MapEcoreOWL* para armazenamento e recuperação dessa informação. Com isso nenhuma restrição ao metamodelos e modelos Ecore teve que ser imposta.

Com a responsabilidade de construir uma *query* em linguagem SparQL e executar uma consulta sobre o modelo OWL a classe *Finder* foi criada. A operação *constructQuery()* é responsável pelo construção da *query*, dado um objeto que a represente o padrão. Após isso *executeQuery()* utiliza os serviços de *WrapperOntologyQueryImpl* para executar a *query* e retornar os resultados do casamento(Figura 4.14).

```
operation execute(pathOntology : String , modelPattern : Object): String is
    ...
    query : String init
    extern wrapper::ontology::management::WrapperOntologyQueryImpl.
    executeQuery(pathOntology,query)
    ...
    result := query
end

operation constructQuery(obj : Object): String is do
constructObjectRelations(obj)
result:= " SELECT " +objects_var+ " \n WHERE {    " +
        relations_var+
        "\n  }"
end

operation constructObjectRelations(obj : Object) : Void is do
...
 // constrói as strings objects_var e relations_var com base no padrão
...
end
```

Figura 4.14 - Construção e execução da query com base no padrão

Por fim, *SearcPattern* utiliza os serviços de *KEcore2OWLConversor*, *InstanceConversor*, *Finder* para prover casamento arquitetural semântico dado um metamodelo qualquer, um modelo válido e padrão arquitetural. Como resultado do casamento de padrão temos um conjunto de mapeamentos entre as variáveis da consulta e os elementos que constituem um modelo, representando todos os possíveis pontos de casamento, sendo *ResultIterator* responsável por prover uma forma de iteração nesses mapeamentos.

As classes e suas funcionalidades apresentadas nos dois projetos foram testadas com o metamodelo UML-RT, modelos e padrões da linguagem. O projeto conta também com uma aplicação introdutória simples do casamento de padrão em um metamodelo para a linguagem de descrição de maquinas de estado.

4.6 Exemplo de uso

Transformações podem ser aplicadas em um dado modelo se este satisfizer a condição necessária de possuir os elementos e relações que satisfaçam o *templete* fonte da transformação. A Figura 4.15, exibi o *template* para a transformação composta decomposição paralela de uma cápsula (lado esquerdo e direito). A partir do *template* foi possível criar um padrão de busca de elementos em modelos UML-RT, onde fosse possível a aplicação da lei (da esquerda para a direita). O padrão casará com todas as cápsulas tipo *BehaviouralCapsule* que estejam em qualquer pacote e possuam pelo menos dois métodos e dois atributos.

```
<metamodel:Model>
  <Packages>
    <Entities xsi:type="metamodel:BehaviouralCapsule">
      <Methods/>
      <Methods/>
      <Attributes/>
      <Attributes/>
    </Entities>
  </Packages>
</metamodel:Model>
```

Figura 4.15 - Padrão de busca decomposição paralela de uma cápsula

Antes de prosseguir com a busca de padrão é necessário converter o modelo em que se realizará a busca, bem como o metamodelo que o descreve no formato OWL.

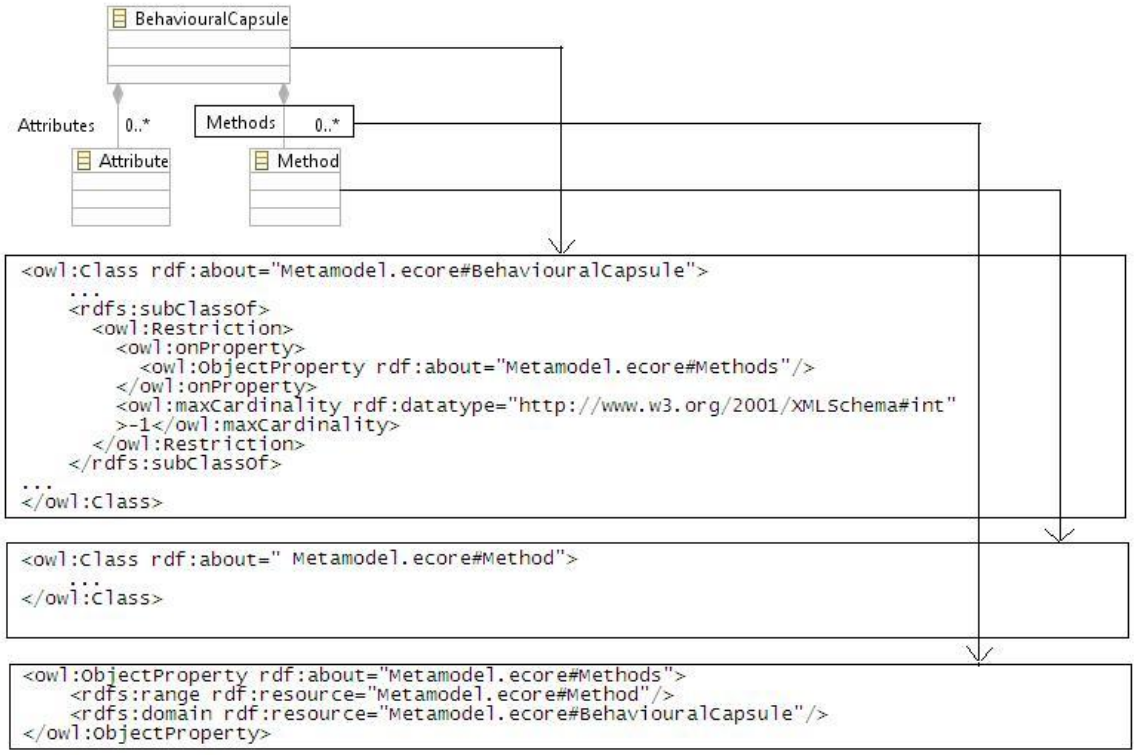


Figura 4.16 - Mapeamento metamodelo Ecore em OWL

A Figura 4.16 mostra parte dessa conversão. O elemento *BehaviourCapsule* é mapeado na classe OWL *"Metamodel.ecore#BehaviouralCapsule"*, que define os identificadores das propriedades (por exemplo, *"Metamodel.ecore#Methods"*) e suas cardinalidades(*-1* indica 0 ou mais elementos). O tipo de elemento de cada relação é definido em seguida nas *tags* <*owl:ObjectProperty*> e <*owl:DataProperty*>, que tem escopo global (por isso a necessidade de todas as propriedades terem identificadores diferentes em OWL).

Na Figura 4.17 é apresentada parte da conversão de um modelo XMI usado para OWL. Quando lido um modelo XMI é representado, em tempo de execução, por objetos Kermeta que possuem um atributo *oid*, que os identifica unicamente, e este é usado com o mesmo propósito para representar os elementos do modelo no formato em OWL.

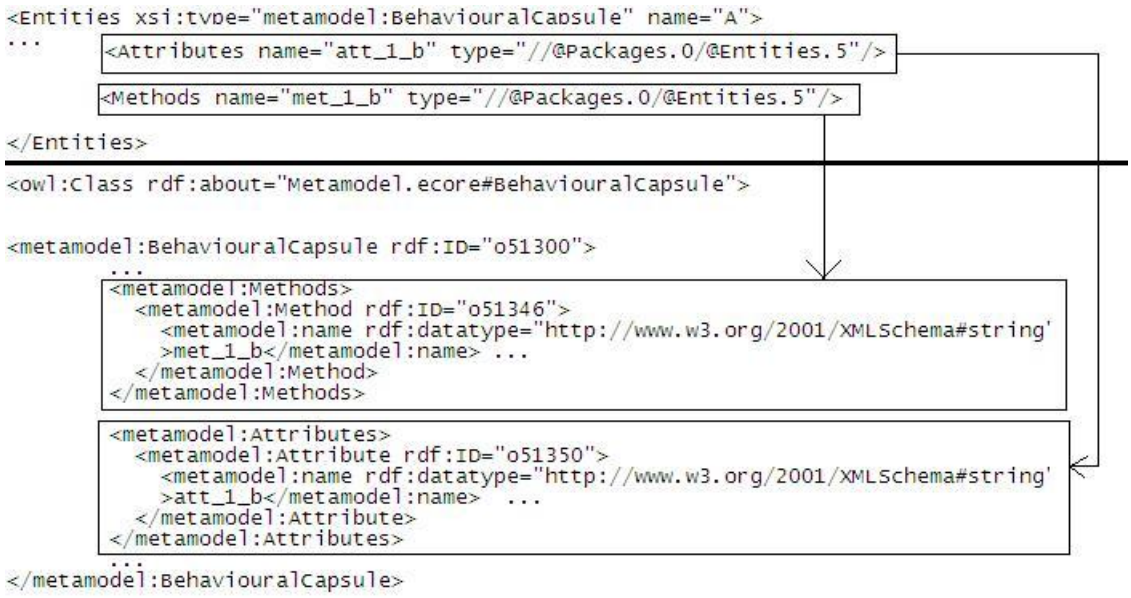


Figura 4.17 - Mapeamento de modelo XMI em OWL

O próximo passo é a transformação do padrão da Figura 4.15 em uma consulta SparQL mostrada na Figura 4.18.

```
//variáveis
SELECT     ?69362 ?69363 ?69364 ?69367 ?69368 ?69365 ?69366

WHERE {
//relações entre elas
 ?69364 metamodel:Attributes  ?69367.
?69364 metamodel:Attributes  ?69368.
?69364 metamodel:Methods  ?69365.
?69364 metamodel:Methods  ?69366.
?69363 metamodel:Entities  ?69364.
?69362 metamodel:Packages  ?69363

 }
```

Figura 4.18 - Consulta SparQL

Quando um padrão é executado o mesmo retorna conjuntos de listas de *ois´s* (uma lista para cada casamento no modelo). Em seguida, cada lista é mapeada em objetos que representam elementos do modelo.

Por fim uma dessas listas randomicamente é escolhida (simulação da escolha por um usuário) e enviada como entrada da transformação, neste caso a decomposição paralela de uma cápsula.

Foram executados testes similares para todas as transformações encapsular cápsula e extrair classe que se encontram no pacote *exampleULM-RT* do projeto *pattermatching.frontend*.

5 Conclusões

Neste trabalho apresentamos uma implementação para um conjunto de *refactorings* (transformações) arquiteturais para modelos expressos em UML-RT [31]. Este conjunto pode ser dividido entre transformações simples e compostas. As primeiras se caracterizam por efetuarem pequenas alterações no modelo, geralmente pela declaração, remoção ou alteração de um único elemento ou propriedade e são utilizadas para justificar as leis compostas, de modo que estas são formadas pela aplicação sucessiva de leis simples.

Para cada conjunto, elaboramos uma estratégia de implementação. A estratégia para implementação das leis simples é baseado no padrão *Command*, que também foi utilizado para as compostas, em adição ao padrão *Iterator*. Cada lei tem seu comportamento representado por um tipo de comando, que provê a cada lei o seu comportamento, que nas leis compostas, está dividido entre o conjunto das leis simples que a justificam $\{L_1,L_2,..L_n\}$. Cada lei simples L_i deve ser aplicada para gerar como saída à entrada de L_{i+1}.

O padrão *Iterator* foi utilizado para constituir o comando e parâmetros corretos a cada iteração.

Como linguagem de implementação das leis optou-se por Kermeta [13] ,dado a sua integração e compatibilidade com as tecnologias Ecore e com a plataforma Java, que possibilitaram tanto a implementação das leis como a definição da API de casamento arquitetural semântico.

A estratégia de implementação de leis é suficientemente genérica e pode ser aplicada ou estendida para outras transformações e possivelmente outras linguagens de modelagem. Além da definição e execução de uma estratégia de automação de leis simples e compostas, se caracteriza como uma contribuição a definição de um metamodelo para UML-RT como uma extensão a [59].

A aplicação de transformações exige que se busque manualmente pelos pontos no modelo onde é possível efetuar a transformação. Para automatizar esse processo, desenvolvemos neste trabalho uma API de casamento de padrões arquiteturais, que é a maior contribuição deste trabalho. Em linhas gerais dada uma tupla (*M,m,P*), onde *M ,n e P* representam respectivamente um metamodelo, um modelo e um padrão arquitetural a API fornece (como seu principal serviço) uma lista L_p ,de todos os pontos no modelo *m*, que casam com o padrão *P*.

O serviço principal da API foi desenvolvido em três etapas: conversão de *M* e *m* para a linguagem de representação de conhecimento(ontologias) OWL [52], conversão de um padrão *P* para a linguagem a linguagem de consulta OWL, SparQL [53], e por fim a execução e tratamento do resultado da consulta.

Para transformação de metamodelos *M* em ontologias utilizou-se os serviços providos pela API EODM (IBM) [54] e os serviços da API Protégé para conversão das instâncias XMI (formato de representação dos modelos) em instâncias OWL. Essa última ainda foi utilizada na geração da *query* de consulta com base num dado padrão *P*.

A construção da API se deu sobre bases tecnológicas sólidas definidas sobre padrões e especificações estáveis e bastante difundidas: MOF da OMG (na forma de sua implementação Ecore) e OWL (SparQL) da W3C em conjunto com a API de projeto Protégé [55]. Esse trabalho contribui (através de um de seus subprodutos) com a adição de algo até então não encontrado na literatura corrente: um intercambio automático entre quaisquer modelos (com base num arbitrário metamodelo *M* conforme a especificação MOF) em instâncias de uma ontologia OWL com semântica equivalente a *M* ou consultas SparQL sobre uma base de modelos.

5.1 Trabalhos relacionados

Uma série de trabalhos tem abordado transformações de modelos, com enfoque no subconjunto destas que preservam o comportamento dos modelos após sua aplicação (*refactorings*) [28, 29, 30].

A maioria dos sistemas de transformação as define para modelos UML [29, 30]. Em uma menor parcela dos trabalhos opta-se pela definição de transformações em modelos escritos em uma linguagem própria ligada a linguagem em que as transformações são implementadas [9, 28]. Esta segunda abordagem implica necessariamente em impor ao usuário o aprendizado de uma nova linguagem, que eventualmente pode ter uma curva de

aprendizado alta. Na primeira abordagem uma linguagem mais geral e estável, como UML, é utilizada na representação dos modelos o que pode ocasionar perda do poder de expressão dos modelos, já que muitos conceitos de arquitetura dirigida a componentes não são representados. Neste trabalho optamos por adotar uma a linguagem UML-RT [31], que estende UML adicionando os conceitos de arquitetura dirigida a modelos.

A aplicação de *refactorings* geralmente está associada à busca pela satisfação de requisitos não funcionais ou *soft-goals* (por exemplo, manutenabilidade) [32]. Em [30], *soft-goals* são utilizados como critério para determinar se transformações devem ser aplicadas em um dado modelo. De modo semelhante a [28] focamos neste trabalho na definição de um conjunto de transformações básicas, que podem ser combinadas progressivamente na definição de transformações cada vez mais complexas para arquiteturas baseadas em componentes.

Uma abordagem comum na literatura para casamento de padrões na atividade de modelagem é baseada na teoria dos grafos [35, 36]. Nessa abordagem metamodelos, modelos e padrões são transformados em grafos o que reduz o problema a uma busca em grafos, o qual possui uma vasta teoria de suporte. Outra forma de prover casamento de padrões e através da representação dos modelos em uma base de dados onde padrões podem ser representados como consultas. Em [37] foi usado o SGBD MySQL para armazenar os modelos, e a linguagem SQL para representar os padrões de consulta. Essa abordagem mostrou-se mais eficiente do ponto de vista de tempo de computação que uma das abordagens mais populares de transformação baseada em garfos, a AGG [36] com base numa bateria de *benchmarks* [37, 38].

Diferentemente das abordagens anteriores, que exigem a definição de padrões numa sintaxe abstrata, neste trabalho padrões são definidos utilizando uma sintaxe concreta cujo metamodelo deriva do utilizado para definir o modelo de entrada da busca. Duas abordagens similares à adotada neste trabalho podem ser encontradas em [39, 40]. A vantagem dessa abordagem reside no fato que a linguagem de definição de padrões é bastante similar à linguagem de definição dos modelos, o que elimina a necessidade de aprendizado de uma nova linguagem para definir padrões.

A utilização de OWL e SparQL confere à API desenvolvida neste trabalho um grau de portabilidade maior que o encontrado em [41, 40]. A manipulação de metamodelos, modelos e consultas em OWL tem sua execução sob a JVM [42], enquanto as outras abordagens têm como pré-requisito a instalação de *engines* com portabilidade restrita como XSB [43] e Flora-2 [44].

5.2 Trabalhos futuros

As leis apresentadas na Tabela 3.1 constituem um subconjunto relevante das leis definidas em [45], a implementação do restante das leis não abordadas neste trabalho podem confirmar a estabilidade da arquitetura e do metamodelo propostos. Há também a necessidade da criação de modelos de teste mais extensos e em maior número como forma de aumentar a garantia sobre a corretude da implementação das leis.

A criação dos modelos apresentados neste trabalho se dá unicamente através de *scripts* Kermeta ou pela edição manual de arquivos XMI. A construção de uma

ferramenta gráfica para a definição e visualização de modelos facilitaria a utilização prática das transformações implementadas.

A adição de novas transformações está condicionada, neste trabalho ao domínio da linguagem Kermeta e extensão do código fonte. Isso dificulta a criação de novas leis, ou customização das existentes, por parte dos usuários finais. Assim uma extensão do trabalho desenvolvido é a definição de uma ferramenta que permita a criação/edição de transformações, abstraindo a linguagem de implementação.

A definição de padrões de busca para aplicação de uma transformação é feita de forma manual. Em um trabalho futuro essa informação poderia ser extraída de forma automática a partir da definição visual das transformações.

Um dos mais importantes trabalhos futuros diz respeito à integração das transformações implementadas neste trabalho, juntamente com a API de casamento de padrões sob a forma de *plugin* em um ambiente de suporte à modelagem.

6 Referências

1. D. Varró, G. Varró, and A. Pataricza, "Designing the Automatic Transformation of Visual Languages," Science of Computer Programming 44, No. 2, 205–227 (2002).
2. D. H. Akehurst and S. J. H. Kent, "A Relational Approach to Defining Transformations in a Metamodel", Proceedings of the 5th International Conference on the Unified Modeling Language, Dresden, Germany (2002), pp. 243–258.
3. A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood, "Transformation: The Missing Link of MDA," Proceedings of the 1st International Conference on Graph Transformation, Barcelona, Spain (2002), pp. 90–105.
4. A. Agrawal, G. Karsai, and F. Shi, "Graph Transformations on Domain-Specific Models", Technical Report ISIS-03-403, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37203 (2003).
5. J. Bézivin, G. Dupé, F. Jouault, G. Pitette, and J. E. Rougui, "First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery," Proceedings of the Workshop on Generative Techniques in the Context of Model Driven Architecture, Anaheim, CA (2003).
6. E. D. Willink, "UMLX: A Graphical Transformation Language for MDA," Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Anaheim, CA (2003), pp. 13–24 (2003).
7. J. de Lara and H. Vangheluwe, "AToM: A Tool for Multi-Formalism and Meta-Modeling," Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering, Grenoble, France (2002), pp. 174–188.
8. P. Braun and F. Marschall, "The Bi-directional Object-Oriented Transformation Language", Technical Report TUM-I0307, Technische Universität München 85748, München, Germany (May 2003).
9. A. Kalnins, J. Barzdins, and E. Celms, "Model Transformation Language MOLA," Proceedings of Model Driven Architecture: Foundations and Applications, Linkoeping, Sweden (2004), pp. 14–28.

10. G. Taentzer, “AGG: A Graph Transformation Environment for Modeling and Validation of Software,” Application of Graph Transformations with Industrial Relevance (AGTIVE'03) 3062, pp. 446–453 (2003).
11. J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez, “Modeling in the Large and Modeling in the Small,” Proceedings of the European MDA Workshops: Foundations and Applications, Twente, The Netherlands (2003), and Linkoeping, Sweden (2004), pp. 33–46.
12. A. Königs, “Model Transformation with Triple Graph Grammars,” Proceedings of Model Transformations in Practice Workshop at MoDELS Conference, Montego Bay, Jamaica (2005)
13. D. Vojtisek and J.-M. Jézéquel, “MTL and Umlaut NG: Engine and Framework for Model Transformation,” http://www.ercim.org/publication/Ercim_News/enw58/vojtisek.html, , acessado em 29 de novembro de 2008.
14. O. Patrascoiu, “YATL: Yet Another Transformation Language,” Proceedings of the 1st European MDA Workshop, Twente, The Netherlands (2004), pp. 83–90.
15. P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, “Weaving Executability into Object-Oriented Metalanguages,” ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica (2005), pp. 264–278.
16. J. Gray, Y. Lin, and J. Zhang, “Automating Change Evolution in Model-Driven Engineering,” IEEE Computer (Special issue on Model-Driven Engineering) **36**, No. 2, 51–58 (February 2006), http://www.cis.uab.edu/gray/Research/C-SAW.
17. L. Tratt, “The MT Model Transformation Language,” Proceedings of ACM SIGAPP Symposium on Applied Computing, Dijon, France (2006).
18. http://www.w3.org/TR/xslt, acessado em 29 de novembro de 2008.
19. SmartQVT documentation, acessado em 29 de novembro de 2008.
20. http://projects.ikv.de/qvt, acessado em 29 de novembro de 2008.
21. http://www.eclipse.org, acessado em 29 de novembro de 2008.
22. http://www.omg.org/technology/documents/formal/xmi.htm, acessado em 29 de novembro de 2008.
23. http://www.cs.ucla.edu/~awarth/papers/dls07.pdf, acessado em 29 de novembro de 2008.
24. M. Kifer, G. Lausen, and J. Wu, “Logical foundations of object-oriented and frame-based languages”. Journal of the ACM, 42:741–843, July 1995.
25. W. Chen, M. Kifer, and D.S.Warren, “HiLog: A foundation for higher-order logic programming.Journal of Logic Programming”, 15(3):187–230, February 1993.
26. A.J. Bonner and M. Kifer. “An overview of transaction logic”. Theoretical Computer Science,133:205–265, October 1994
27. A.J. Bonner and M. Kifer. “A logic for programming database transactions”. In J. Chomicki and G. Saake, editors, Logics for Databases and Information Systems, chapter 5, pages 117–166.Kluwer Academic Publishers, March 1998.
28. Grunske, L. “Formalizing Architectural Refactorings as Graph Transformation Systems”. In the Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 05), IEEE Computer Society vol. 6, 2005, 324 – 329.

29. Hosseini, S. and Azgomi, M. A. “UML Model Refactoring with Emphasis on Behavior Preservation”. Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering - Volume 00
30. Ivkovicand, I. and Kontogiannis, K.. “A Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations”, Proceedings of the Conference on Software Maintenance and Reengineering,2006
31. B. Selic and J. Rumbaugh. “Using UML for Modeling Complex RealTime Systems”. Rational Software Corporation, 1998. Disponível em http://www. rational.com.,acessado em 29 de novembro de 2008
32. Chung, L., Nixon, B., Yu, E. And Mylopoulos, J. “Non-Functional Requirements in Software Engineering”. Kluwer Publishing, 2000.
33. M. Fowler. “Refactoring: Improving the Design of Existing Code”. Addison-Wesley, Upper Saddle River, NJ, 2000.
34. Kalnins, A., Celms, E., Sostaks, A.: “Simple and efficient implementation of pattern matching in mola tool”. In: Baltic DB&IS2006, Vilnius, Lithuania (2006) 159–167
35. T. Levendovszky, L. Lengyel, G. Mezei, H. Charaf, “A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS”, ENTCS, International Workshop on Graph-Based Tools, GraBaTs, Rome, 2004
36. AGG – “The Attributed Graph Grammar System”. Disponível em http://tfs.cs.tu-berlin.de/agg/,acessado em 29 de novembro de 2008
37. Kalnins, A., Celms, E., Sostaks, A.: Simple and efficient implementation of pattern matching in mola tool. In: Baltic DB&IS2006, Vilnius, Lithuania (2006) 159–167
38. G. Varro, A. Schurr, and D. Varro, “Benchmarking for graph transformation,” in Proc. IEEE Symposium on Visual Languages and Human-Centric Computing 2005 (VL/HCC 05), 2005, pp 79-88.
39. Baar, T., Whittle, J.: “On the usage of concrete syntax in model transformation rules”. In: 6th International Andrei Ershov Memorial Conference Perspectives of System Informatics. LNCS, Springer (2006)
40. R Ramos, O Barais, JM Jezequel, “Matching Model-Snippets,” in Lecture Notes in Computer Science, 2007 - Berlin: Springer-Verlag, 1973
41. Prechelt, L., Kramer, C.: “Functionality versus practicality: Employing existing tools for recovering structural design patterns”. Journal of Universal Computer Science (J.UCS) 4 (1998) 866–882
42. http://www.sun.com/, acessado em 30 de novembro de 2008
43. K. Sagonas T. Swift and D. S. Warren (1994). “XSB as an Efficient Deductive Database Engine”. In Snodgrass RT, Winslett M (eds.) 1994 ACM SIGMOD Internacional Conference on Management of Data (SIGMOD 94), pp 442-453.
44. G. Yang, M. Kifer, and C. Zhao. Flora-2: User's Manual. http://flora.sourceforge.net/, acessado em 30 de novembro de 2008
45. RAMOS, R. “Desenvolvimento Rigoroso com UML-RT”. Dissertação de mestrado, Universidade Federal de Pernambuco, Centro de Informática, 2005.
46. D.M. Coleman, D. Ash, B. Lowther, and P.W. Oman, “Using Metrics to Evaluate Software System Maintainability,” Computer,vol. 27, no. 8, pp. 44-49, Aug. 1994.

47. T. Guimaraes, “Managing Application Program Maintenance Expenditure,” Comm. ACM, vol. 26, no. 10, pp. 739-746, 1983.
48. B.P. Lientz and E.B. Swanson, “Software Maintenance Management: A Study of the Maintenance of Computer Application Software” in 487 Data Processing Organizations. Addison-Wesley, 1980.
49. Anneke Kleppe, Jos Warmer, Wim Bast, “MDA Explained, The Model Driven Architecture:Practise and Promise”, 2003, Addison-Wesley
50. W.F. Opdyke, “Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks,” PhD thesis,Univ. of Illinois at Urbana-Champaign, 1992.
51. http://www.eclipse.org/modeling/emf/, acessado em 29 de novembro de 2008.
52. Deborah L. McGuinness and Frank van Harmelen. “Owl web ontology language overview”. http://www.w3.org/TR/owl-features/, 2008.
53. E. Prud’hommeaux and A. Seaborne. “SPARQL Query Language for RDF”. http://www.w3.org/TR/rdf-sparql-query/, Janeiro de 2008.
54. EODM – EMF Ontology Definition Metamodel. http://www.eclipse.org/modeling/mdt/?project=eodm, acessado em 29 de novembro de 2008.
55. Holger Knublauch. “Protégé-owl API programmer's guide”. http://protege.stanford.edu/plugins/owl/api/guide.html. setembro de 2006, acessado em 29 de novembro de 2008.
56. Tom Mens, Tom Tourwé, "A Survey of Software Refactoring," IEEE Transactions on Software Engineering, vol. 30, no. 2, pp. 126-139, February, 2004.
57. Frankel, D.S., “Model Driven Architecture: Applying MDA to Enterprise Computing”. 2003: OMG Press (Wiley Publishing, Inc.).
58. E.J. Chikofsky and J.H. Cross, “Reverse Engineering and Design Recovery: A Taxonomy,” IEEE Software, vol. 7, no. 1, pp. 13-17,1990.
59. Lima, E. “Automatizando Refatoramentos Arquiteturais em UML-RT utilizando Transformação de Modelos”. Trabalho de Graduação, Universidade Federal de Pernambuco, Centro de Informática, 2008.
60. http://www.omg.org/technology/documents/formal/xmi.htm, acessado em 29 de novembro de 2008.
61. Stephen J. Mellor, Anthony N. Clark, Takao Futagami, "Guest Editors' Introduction: Model-Driven Development," *IEEE Software*, vol. 20, no. 5, pp. 14-18, September/October, 2003.
62. http://www.omg.org/, acessado em 29 de novembro de 2008.
63. http://www.corba.org/, acessado em 29 de novembro de 2008.
64. http://www.uml.org/, acessado em 29 de novembro de 2008.
65. http://www.omg.org/mof/, acessado em 29 de novembro de 2008.
66. Mukerji, J., Miller, J.: “MDA Guide”. http://www.omg.org/docs/omg/03-06-01.pdf. (2008)
67. B. Selic, G. Gullekson, and P. T. Ward. Real-time object-oriented modeling. John Wiley & Sons, Inc., 1994.
68. http://www.w3.org/TR/xquery/, acessado em 29 de novembro de 2008.

69. Interactive Software Engineering Inc.. “Eiffel: The Language,Version 2.2”. Santa Barbara, CA, USA, 1989.
70. http://www.w3.org/, acessado em 29 de novembro de 2008.

Orientador

_____

Prof. Ph.D. Augusto Cesar Alves Sampaio

Aluno

_____

José Dihego da Silva Oliveira