COPPE/UFRJ

ODYSSEY-MEC: UMA ABORDAGEM PARA O CONTROLE DA EVOLUÇÃO DE MODELOS COMPUTACIONAIS NO CONTEXTO DO DESENVOLVIMENTO DIRIGIDO POR MODELOS

Chessman Kennedy Faria Corrêa

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Cláudia Maria Lima Werner
Leonardo Gresta Paulino Murta

Rio de Janeiro
Fevereiro de 2009

ODYSSEY-MEC: UMA ABORDAGEM PARA O CONTROLE DA EVOLUÇÃO DE MODELOS COMPUTACIONAIS NO CONTEXTO DO DESENVOLVIMENTO DIRIGIDO POR MODELOS

Chessman Kennedy Faria Corrêa

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof.ª Cláudia Maria Lima Werner, D.Sc.

Prof. Leonardo Gresta Paulino Murta, D.Sc.

Prof. Geraldo Bonorino Xexéo, D.Sc.

Prof. Eber Assis Schmitz, Ph.D.

RIO DE JANEIRO, RJ – BRASIL

FEVEREIRO DE 2009

Corrêa, Chessman Kennedy Faria

Odyssey-MEC: Uma Abordagem para o Controle da Evolução de Modelos Computacionais no Contexto do Desenvolvimento Dirigido por Modelos / Chessman Kennedy Faria Corrêa. – Rio de Janeiro: UFRJ/COPPE, 2009

XIV, 122 p.: il, 29,7 cm

Orientadores: Cláudia Maria Lima Werner

Leonardo Gresta Paulino Murta

Dissertação (mestrado) – UFRJ/ COPPE / Programa de Engenharia de Sistemas e Computação, 2009

Referências bibliográficas: p 116 a 122.

1. Desenvolvimento Dirigido por Modelos. 2. Arquitetura Dirigida por Modelos. 3. Transformação de Modelos. 4. Sincronização de Modelos. 5. Evolução de Modelos. I. Werner, Cláudia Maria Lima *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III Título.

A meus pais Josélia e Jorge Corrêa

À minha esposa Adriana

Agradecimentos

A Deus.

À minha esposa Adriana, por ter sido paciente e compreensiva, permitindo que eu pudesse prosseguir com tranqüilidade, e pela preocupação demonstrada dia a dia.

À professora Cláudia Maria Lima Werner, por ter me dado a chance de realizar este tão esperado sonho, ter contribuído para a minha formação e possibilitado a oportunidade de adquirir novas experiências no campo da ciência.

Ao professor Leonardo Murta, por todo o seu auxílio nesta jornada, pela orientação paciente, pelo incentivo nos momentos de cansaço e, antes de tudo, por ter sido um amigo.

Aos amigos do Laboratório de Engenharia de Software pelos momentos agradáveis que passamos juntos nas conferências.

Aos professores Geraldo Bonorino Xexéo e Eber Assis Schmitz por terem aceitado o convite para participar desta banca, agregando valor ao trabalho realizado.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

ODYSSEY-MEC: UMA ABORDAGEM PARA O CONTROLE DA EVOLUÇÃO DE MODELOS COMPUTACIONAIS NO CONTEXTO DO DESENVOLVIMENTO DIRIGIDO POR MODELOS

Chessman Kennedy Faria Corrêa

Fevereiro/2009

Orientadores: Cláudia Maria Lima Werner
Leonardo Gresta Paulino Murta

Programa: Engenharia de Sistemas e Computação

O Desenvolvimento Dirigido por Modelos (MDD – *Model Driven Development*) utiliza modelos como principais artefatos para o desenvolvimento de software. Nesta abordagem, os modelos são refinados sucessivamente através de transformações. O objetivo é criar um modelo que seja suficientemente completo para a geração automática de código fonte e de outros tipos de artefatos com formato texto. Contudo, os diferentes modelos podem evoluir de maneira independente, tornando-se inconsistentes ao longo do tempo, principalmente quando são manipulados por profissionais distintos ou quando ferramentas CASE diferentes são usadas.

Este trabalho apresenta o Odyssey-MEC (*Odyssey for Model Evolution Control),* uma abordagem para fazer o controle de evolução de modelos inter-relacionados de projetos MDD ao longo do tempo e do espaço. A evolução dos modelos no tempo é controlada a partir do versionamento de cada modelo, enquanto a evolução no espaço é realizada através de sincronização, o que mantém os modelos inter-relacionados consistentes entre si. A aplicação da abordagem é viabilizada a partir de um protótipo constituído de uma arquitetura cliente-servidor, de modo que as transformações, sincronização e versionamento são realizadas no lado servidor.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

ODYSSEY-MEC: AN APPROACH FOR COMPUTATIONAL MODEL EVOLUTION CONTROL IN THE CONTEXT OF MODEL DRIVEN DEVELOPMENT

Chessman Kennedy Faria Corrêa

February/2009

Adivisors: Cláudia Maria Lima Werner
Leonardo Gresta Paulino Murta

Department: Computing and Systems Engineering

Model Driven Development (MDD) uses models as main artifacts to software development. In this approach, models are refined through transformations. The objective is to create a complete model to automatically generate source code and other text based artifacts. However, different models may evolve independently, becoming inconsistent along the time, particularly when different professionals manipulate the models or different CASE tools are used.

This work presents Odyssey-MEC (Odyssey for Model Evolution Control), an approach to control MDD inter-related models evolution during time and space. Time model evolution is controlled by model versioning, and space model evolution is controlled by model synchronization, which maintains inter-related models consistent with each other. The approach application is enabled by a prototype based on a client-server architecture, where transformation, synchronization and versioning are executed at the server side.

Índice

Capítulo 1 – Introdução 1

1.1 Motivação 1

1.2 O Problema 3

1.3 Objetivos 4

1.4 Organização 5

Capítulo 2 – Controle de Evolução de Modelos 6

2.1 Introdução 6

2.2 Arquitetura Dirigida por Modelos (MDA) 8

2.2.1 Modelos MDA 8

2.2.2 Linguagem para Representação de Modelos 9

2.2.3 Transformação de Modelos 10

2.2.4 Registro de Transformação 13

2.2.5 Níveis de Abstração de Modelos 15

2.2.6 O Metadata Object Facility (MOF) 16

2.2.7 Interoperabilidade de Modelos 16

2.2.8 Aplicação do MDA no Desenvolvimento Dirigido por Modelos 18

2.3 Consistência entre Modelos 19

2.3.1 Propriedades da Sincronização 21

2.3.2 Localização de Elementos 22

2.3.3 Quantidade de Elementos 22

2.3.4 Propagação de Modificações 22

2.4 Consistência Intra-Modelo 23

2.5 Gerência de Configuração e Controle de Versão 24

2.5.1 Item de Configuração, Versão e Configuração de Software 24

2.5.2 Sistemas de Controle de Versão 25

2.5.3 Políticas para Controle de Concorrência 26

2.5.4 Controle de Versão de Modelos 27

2.6 Considerações Finais 28

Capítulo 3 – Abordagens MDD Relacionadas com a Evolução de Modelos 29

3.1 Introdução 29

3.2 Critérios para a Comparação das Abordagens 29
3.3 Abordagens Existentes na Literatura 31
3.3.1 C-SAW 31
3.3.2 Abordagem Proposta por Xiong et al 32
3.3.3 SMoVER 33
3.3.4 Abordagem Proposta por Alanen 34
3.3.5 Abordagem Proposta por Sriplakich et al 35
3.3.6 Odyssey-VCS 35
3.3.7 Odyssey-MDA 36
3.3.8 Together 36
3.3.9 Enterprice Architect 37
3.3.10 Outras Abordagens 37
3.4 Análise Comparativa das Abordagens 39
3.5 Considerações Finais 40
Capítulo 4 – Odyssey-MEC: *A* Abordagem Proposta 42
4.1 Introdução 42
4.2 Visão Geral da Abordagem 43
4.3 Definição de Mapas de Transformação de Modelos 45
4.3.1 Definição de Critérios de Busca 47
4.3.2 Definição de Mecanismos de Transformação 47
4.3.3 Definição de Criação de Relacionamentos 48
4.3.4 Mapa de Transformação para Tipos Simples 49
4.3.5 Mapas de Transformação no Odyssey-MEC 50
4.4 Configuração de Projetos MDD 50
4.5 Marcação de Modelos 52
4.6 Transformação Automática de Modelos 52
4.6.1 Não Transformação de Elementos 53
4.7 Ligações de Rastreabilidade 53
4.8 Controle de Versões de Modelos 54
4.8.1 Identificação e Localização de Elementos 57
4.8.2 Junção de Modelos 58
4.8.3 Recuperação de Perfis Aplicados aos Modelos 60
4.9 Sincronização de Modelos 60
4.9.1 Recuperação de Dados de Versionamento 62

4.9.2 Localização de Elemento 64
4.9.3 Recuperação de Elementos Específicos do Modelo 66
4.9.4 Propagação da Sincronização 66
4.10 Compartilhamento de Modelos 67
4.11 Integridade dos Modelos 68
4.12 Considerações Finais 68
Capítulo 5 – Exemplos de Aplicação da Abordagem 72
5.1 Introdução 72
5.2 Inclusão de um Elemento Transformável no PIM 73
5.3 Alteração de um Elemento Transformável no PIM 74
5.4 Exclusão de um Elemento Transformável do PIM 75
5.5 Inclusão de um Elemento Transformável no PSM 75
5.6 Alteração de um Elemento Transformável do PSM 76
5.7 Exclusão de um Elemento Transformável do PSM 76
5.8 Inclusão de um Elemento não Transformável no PSM 77
5.9 Alteração de um Elemento não Transformável do PSM 78
5.10 Exclusão de um Elemento não Transformável do PSM 78
5.11 Conflito a partir da Alteração do Mesmo Modelo 79
5.12 Conflito a partir de Alterações de Modelos Diferentes 79
5.13 Considerações Finais 79
Capítulo 6 – Protótipo do Odysssey-MEC 81
6.1 Introdução 81
6.2 Contexto de Uso 81
6.2.1 Eclipse como Ferramenta de Modelagem 82
6.2.2 Utilização do Odyssey-MEC 83
6.3 Detalhamento da Implementação 84
6.3.1 Repositório de Modelos 86
6.3.2 Versionamento de Modelos 87
6.3.3 Transformação de Modelos 90
6.3.4 Sincronização de Modelos 92
6.3.5 Controle de Transação 96
6.3.6 Comunicação Cliente-Servidor 97
6.4 Uso do Protótipo 98
6.4.1 Preparação do Projeto 99

6.4.2 Realizando o Login 100
6.4.3 Importação de um Modelo para o Cliente do Odyssey-MEC 101
6.4.4 Marcação de Modelos 101
6.4.5 Check-in de um Modelo 102
6.4.6 Check-out de um Modelo 103
6.4.7 Exportação de um Modelo 104
6.4.8 Visualização de Vários Modelos 104
6.5 Exemplos de Uso do Protótipo 104
6.5.1 Exemplo 1: Primeiro Check-in do PIM 105
6.5.2 Exemplo 2: Modificação do PIM 106
6.5.3 Exemplo 3: Modificação do PSM-JEE 107
6.5.4 Exemplo 4: Conflito de Alteração entre Modelos Diferentes 109
6.6 Considerações Finais 109
Capítulo 7 – Conclusões 111
7.1 Contribuições 111
7.2 Limitações 112
7.2.1 Uso de Padrões da OMG 112
7.2.2 Dependência de Especificação de Transformações 112
7.2.3 Uso de Apenas um Modelo como Entrada 113
7.2.4 Verificação de Consistência Intra-Modelo 113
7.2.5 Propagação Indireta de Transformações entre PSMs 113
7.2.6 Geração de Modelos de Comportamento 113
7.2.7 Desempenho 114
7.3 Trabalhos Futuros 114
7.3.1 Uso de Regras para Controlar a Evolução dos Modelos 114
7.3.2 Controle da Evolução do CIM 114
7.3.3 Controle da Evolução de Código Fonte 114
7.3.4 Controle de Evolução de Mapas de Transformação 115
7.3.5 Controle de Metamodelos 115
7.3.6 Visualização da Evolução dos Modelos Inter-Relacionados 115
7.3.7 Avaliação da Abordagem 115
Referências Bibliográficas 116

Índice de Figuras

Figura 2.1: Criação de modelos e código no MDD. Adaptado de *FONDEMENT et al.* (2004) 6
Figura 2.2: Processo de transformação do MDA 11
Figura 2.3: Transformação da abordagem traducionista 12
Figura 2.4: Transformação da abordagem elaboracionista 13
Figura 2.5: Registro de Transformação 14
Figura 2.6: Modelos Inter-Relacionados 16
Figura 2.7. Aplicação do MDA no Desenvolvimento Orientado a Modelos 19
Figura 2.8. Propagação de alterações entre modelos diferentes 23
Figura 2.9. Sistema de Controle de Versão 26
Figura 3.1: Algoritmo de sincronização. Adaptado de XIONG *et al.* (2007) 32
Figura 4.1. Abordagem Odyssey-MEC 43
Figura 4.2: Exemplo de mapeamento 46
Figura 4.3. Modelo de transformações do Odyssey-MDA (MAIA, 2006) 46
Figura 4.4: Especificação de critérios de busca (MAIA, 2006) 47
Figura 4.5: Propriedades de configuração do mapeamento de transformação (MAIA, 2006) 48
Figura 4.6: Propagação de relacionamento 48
Figura 4.7: Criação de um relacionamento de generalização 49
Figura 4.8: Geração de um modelo a partir de outro 51
Figura 4.9: Meta-modelo para a especificação de um projeto MDD 51
Figura 4.10: Ligação de rastreabilidade 53
Figura 4.11: Exemplo de variação de granuralidade - Adaptado de OLIVEIRA (2005) 54
Figura 4.12: Meta-modelo do Odyssey-VCS para o controle de versão 56
Figura 4.13: Junção de versões diferentes de um item de configuração 58
Figura 4.14: Recuperação de dados e versionamento 61
Figura 4.15: Relacionamentos no tempo e no espaço 62
Figura 4.16: Procedimento de recuperação de dados de versionamento 63
Figura 4.17: Ligações de rastreabilidade 65
Figura 4.18: Exemplo de grafo de propagação de sincronização 66

Figura 5.1: Modelo de casos de uso de classes (PIM) usado nos exemplos 72
Figura 6.1: Eclipse como ambiente de modelagem 82
Figura 6.2: Processo realizado para o versionamento, transformação e sincronização de modelos 83
Figura 6.3: Arquitetura do Odyssey-MEC 85
Figura 6.4: Gerenciadores de repositório 87
Figura 6.5: Interface genérica de operações de controle de versão 88
Figura 6.6: Visão parcial na nova implementação do Odyssey-VCS 89
Figura 6.7: Estrutura básica do Odyssey-MDA 91
Figura 6.8: Máquina de sincronização e suas dependências 93
Figura 6.9: Execução da máquina de sincronização a partir do *check-in* 94
Figura 6.10: Processo de transformação e sincronização 94
Figura 6.11: Hooks para transformação esquerda-direita e direita-esquerda 95
Figura 6.12: Gerenciador de transações 97
Figura 6.13: Comunicação cliente-servidor do Odyssey-MEC 98
Figura 6.14: *Web Services* do Odyssey-MEC 98
Figura 6.15: Tela principal do Odyssey-MEC Manager 98
Figura 6.16: Tela de especificação do projeto 99
Figura 6.17: Tela de especificação de modelo 99
Figura 6.18: Tela de especificação de transformação 99
Figura 6.19: *Login* em um repositório 100
Figura 6.20: Importação do modelo 100
Figura 6.21: Model Marker (MAIA, 2006) 101
Figura 6.22: Check-in de um modelo (PIM) 102
Figura 6.23: Modelo-fonte versionado após o check-in 102
Figura 6.24: Versão obtida a partir da operação de check-out 103
Figura 6.25: Exportação de um modelo 103
Figura 6.26: Visualização de mais de um repositório 104
Figura 6.27: Modelo de casos de uso de classes (PIM) usado nos exemplos 105
Figura 6.28: PIM e PSMs após o primeiro *check-in* do PIM 106
Figura 6.29: PIM e PSMs após a sincronização 107
Figura 6.30: PIM e PSMs após alteração do PSM-JEE e sincronização 108
Figura 6.31: Conflito provocado por modelos diferentes 109

Índice de Tabelas

Tabela 2.1: Níveis de metadados da OMG. 17
Tabela 3.1: Quadro comparativo das abordagens 40
Tabela 4.1: Propriedades a associações do elemento *Version* 56
Tabela 4.2: Resultados da análise de existência – Adaptado de MURTA (2008) 59
Tabela 4.3: Processamento de atributos – Adaptado de MURTA (2008) 59
Tabela 4.4: Quadro comparativo das abordagens 70
Tabela 4.5: Odyssey-MEC e seus componentes em relação ao tempo e ao espaço 71

Capítulo 1 – Introdução

1.1 Motivação

O desenvolvimento moderno de *software* tem como finalidade a criação de soluções de qualidade para atender às necessidades daqueles que irão utilizá-las (TIAN, 2005). No início, as limitações dos recursos computacionais não permitiam a criação de software sofisticado (SEBESTA, 2005). Assim, os problemas que podiam ser solucionados eram limitados. Com o passar do tempo, a evolução dos recursos computacionais tornou possível a criação de programas de computador maiores e mais complexos (PRESSMAN, 2002). Durante essa evolução, várias linguagens de programação de alto nível foram criadas, afastando a lógica de programação do *hardware* e deixando-a mais próxima do raciocínio humano. Algumas dessas linguagens permitiram a estruturação de programas em módulos de modo que a solução de um problema pudesse ser implementada em partes, tornando o programa mais fácil de entender e manter (MELLOR *et al.*, 2004). A programação orientada a objetos melhorou ainda mais a organização dos programas, mantendo em uma mesma estrutura os dados e as rotinas que podem operá-los (MELLOR *et al.*, 2004).

O tamanho e a complexidade do problema a ser resolvido e da solução a ser implementada fez com que o uso de linguagens de programação de alto nível não fosse suficiente para a criação de *software* com a qualidade esperada e dentro dos custos e prazos estabelecidos (MELLOR *et al.*, 2004). Assim, juntamente com as metodologias de análise e desenho[1], foram criadas técnicas de modelagem que possibilitassem o melhor entendimento e representação do problema e da solução. A UML (*Unified Modeling Language*) (BOOCH *et al.*, 2005) é um exemplo de linguagem de modelagem criada para representar problemas e soluções no contexto do desenvolvimento orientado a objetos. Hoje, é difícil encontrar uma metodologia de desenvolvimento que não utilize técnicas de modelagem para a compreensão do problema e da solução. Assim, o uso de modelos caracteriza uma nova elevação do nível de abstração do desenvolvimento de software (MELLOR *et al.*, 2004).

---

[1] Do inglês *design*. Outra tradução comum é projeto. Contudo, a palavra *desenho* foi escolhida com o objetivo de distinguir esta etapa de desenvolvimento do projeto (*project*) como um todo.

Apesar da modelagem ter se tornado um recurso essencial para o desenvolvimento de software, as metodologias tradicionais de desenvolvimento têm como principal artefato o código fonte. Nestas metodologias, o modelo tem como principais finalidades permitir a compreensão e representação do problema, servir como documentação de alto nível da solução e facilitar a comunicação entre os membros da equipe de desenvolvimento. Neste contexto, o código fonte geralmente evolui e os modelos ficam desatualizados com o tempo; e normalmente permanecem neste estado (BEYDEDA *et al.*, 2005).

O elevado nível de complexidade e tamanho dos programas de computador, além da mudança contínua de tecnologia, fez com que um novo paradigma de desenvolvimento de software fosse criado: o *Model-Driven Development* (MDD) (Desenvolvimento Dirigido por Modelos) (KEPPLE *et al.*, 2002). Nessa abordagem, os modelos são considerados como os principais artefatos do desenvolvimento e, freqüentemente, são usados para a geração automática de código fonte. Desse modo, o nível de abstração definitivamente sobe do código fonte para os modelos.

Durante a sua existência, o software é muitas vezes modificado com o objetivo de atender a novos requisitos funcionais, substituir tecnologia ou corrigir defeitos. Essas alterações caracterizam a evolução do software, e a gerência de configuração é um dos recursos usados para fazer o controle dessa evolução (ESTUBLIER, 2000). Como sub-tarefa, está o controle de versões, a partir do qual é criado um histórico de todas as alterações realizadas. Assim, a evolução do software, particularmente do código fonte, pode ser controlada ao longo do tempo. A partir do histórico de versões, é possível analisar como artefatos evoluem. Além disso, é possível retornar a uma versão anterior do projeto quando algum problema é identificado.

A complexidade, o tamanho do software e a necessidade de reduzir o tempo de construção fez com que o desenvolvimento se tornasse uma atividade realizada por um conjunto de pessoas trabalhando em equipe, podendo, inclusive, estarem geograficamente distribuídas (SANGWAN *et al.*, 2007). Esse fato tornou o controle de evolução de software ainda mais importante.

Assim como nas abordagens de desenvolvimento em que o principal artefato é o código fonte, os mesmos problemas que tornaram necessário o controle da evolução do código surgem no MDD. Contudo, o problema é ainda mais abrangente, pois, além do código fonte, é necessário controlar a evolução de modelos inter-relacionados,

considerando não apenas o versionamento, mas também a consistência entre os artefatos.

1.2 O Problema

Sistemas de controle de versão vêm sendo usados na Gerência de Configuração como recursos para manter a evolução de software sob controle (BERSOFF *et al.*, 1980). Esses sistemas possuem uma arquitetura que permite diversos desenvolvedores trabalharem de qualquer lugar no mesmo projeto de software.

No MDD, o desenvolvimento ocorre a partir do refinamento sucessivo de modelos. Desse modo, um modelo é usado para a geração de outro através de uma transformação, preferencialmente automática. Assim, esses modelos, apesar de independentes, estão inter-relacionados.

Depois de gerados, os modelos podem evoluir independentemente ao longo do tempo. Por exemplo, analistas podem modificar modelos de alto nível em função de novos requisitos, enquanto projetistas podem modificar modelos de baixo nível para acrescentar detalhes para a geração de código fonte.

A evolução independente dos modelos inter-relacionados pode torná-los inconsistentes. A inconsistência é caracterizada pela: (1) ausência de elementos em um modelo que estão relacionados com elementos de outro modelo, decorrente de operações de inclusão e exclusão, e (2) diferença entre as propriedades dos elementos de modelos diferentes, decorrente da alteração dos elementos em um modelo, mas não nos elementos correspondentes dos outros modelos. A relação entre os elementos é decorrente da transformação realizada. Por exemplo, dados os modelos A e B, uma transformação T, que transforma A em B, pode determinar a criação de uma classe CB_1 no modelo B em função da existência de uma classe CA_1 com determinadas características no modelo A. Se uma classe CA_2, com as mesmas características de CA_1, for incluída manualmente em A, os modelos A e B passam a estar inconsistentes devido B não possuir a classe CB_2 correspondente. Do mesmo modo, se a classe CA_1 for excluída de A, os modelos ficam inconsistentes porque B ainda tem a classe CB_1, correspondente a CA_1. Finalmente, a mudança do nome da classe CA_1 para CAA_1, torna os modelos inconsistentes porque a classe correspondente CB_1 não foi alterada para CBB_1.

Os sistemas de controle de versão para modelos (ALANEN, 2002; OLIVEIRA *et al.*, 2005; SMOVER, 2008; SRIPLAKICH *et al.*, 2008) estão focados em manter a

evolução de modelos sem levar em consideração a inter-relação entre modelos diferentes. Desse modo, apesar de ser possível controlar a evolução de modelos ao longo do tempo, não há como garantir a sua consistência.

Para que modelos permaneçam consistentes, é necessário sincronizá-los à medida que são alterados. Contudo, o esforço necessário para a realização manual dessa tarefa pode fazer com que esta não seja realizada. Mesmo que uma ferramenta para a sincronização automática de modelos esteja disponível, é possível que os desenvolvedores, por diversos motivos, esqueçam de executá-la.

Além do problema de manter a consistência de modelos independentes, em um ambiente distribuído de desenvolvimento, é possível que ferramentas CASE diferentes sejam usadas para a manipulação dos modelos inter-relacionados.

1.3 Objetivos

O objetivo deste trabalho é apresentar uma abordagem para controlar a evolução de modelos computacionais inter-relacionados ao longo do desenvolvimento de software que utiliza o paradigma MDD. Neste trabalho, entende-se como evolução as alterações realizadas nos modelos ao longo do tempo em função da correção, atendimento de novos requisitos e mudança nos padrões das tecnologias para as quais os modelos são usados, para a geração de código fonte ou outros artefatos[2]. Neste caso, o controle da evolução está sendo caracterizado como a tarefa executada para garantir que as alterações não conflitantes, realizadas em um mesmo modelo por desenvolvedores diferentes, sejam automaticamente unificadas em um único modelo, deixando para o engenheiro de software apenas a solução de conflitos. Além disso, as alterações em elementos de um modelo que tenham relação com elementos de outros modelos devem ser devidamente propagadas, de modo que os modelos permaneçam consistentes. Finalmente, um controle de evolução de modelos deve possibilitar a recuperação de versões anteriores às modificações.

Com base no problema apresentado, a abordagem proposta nesta dissertação tem os seguintes sub-objetivos:

- Possibilitar que desenvolvedores com papéis diferentes possam trabalhar no mesmo projeto de software.

[2] Modelos também podem ser usados para a geração de outros tipos de artefatos, como, por exemplo, um *script* para a geração das tabelas de um banco de dados.

- Fazer a transformação e a sincronização automática de modelos independentemente da iniciativa do desenvolvedor, de modo que os modelos sejam sempre transformados e permaneçam consistentes.
- Manter as versões de todos os modelos que constituem um projeto de software MDD.
- Possibilitar o uso de ferramentas CASE diferentes.
- Registrar a inter-relação existente entre os modelos decorrente do processo de transformação.

1.4 Organização

Este trabalho está organizado em 6 capítulos, além desta introdução. No segundo capítulo, são ilustrados os principais conceitos sobre o *Desenvolvimento Dirigido por Modelos,* utilizando a *Arquitetura Dirigida por Modelos* como referência. Neste capítulo, também são tratadas a geração, sincronização e versionamento de modelos.

O terceiro capítulo apresenta algumas abordagens para o controle da evolução de modelos e um conjunto de critérios para a comparação dessas abordagens, como, por exemplo, o uso de sincronização, versionamento de modelos, possibilidade de uso de qualquer ferramenta CASE e emprego de formatos padrões definidos pela OMG.

No quarto capítulo, a abordagem proposta é apresentada, sendo constituída de uma estrutura que permite o controle da evolução do software criado a partir do MDD, segundo os critérios especificados no terceiro capítulo.

No quinto capítulo, são apresentados alguns exemplos de aplicação da abordagem.

No sexto capítulo, é descrito o protótipo que implementa a abordagem proposta. São apresentados alguns exemplos de uso do protótipo, tal como a criação de modelos a partir de outro e a sincronização após alterações.

Finalmente, no sétimo capítulo, são apresentados as contribuições, limitações e os trabalhos futuros.

Capítulo 2 – Controle de Evolução de Modelos

2.1 Introdução

O **Desenvolvimento Dirigido por Modelos** (**MDD** – *Model-Driven Development*) (SOLEY, 2000) é caracterizado pelo uso de modelos como principais artefatos no processo de desenvolvimento de software. Nessa abordagem, um modelo é usado para a geração de outro. Esses modelos podem estar no mesmo nível de abstração ou em níveis de abstração diferentes. Modelos em níveis de abstração mais altos estão mais afastados das particularidades de uma plataforma de software, enquanto modelos menos abstratos estão mais próximos das especificações da plataforma (Figura 2.1). Independentemente do número de modelos que são criados durante o desenvolvimento e dos seus respectivos níveis de abstração, o objetivo final é gerar o código fonte do programa para uma plataforma de software particular (FONDEMENT *et al.*, 2004).

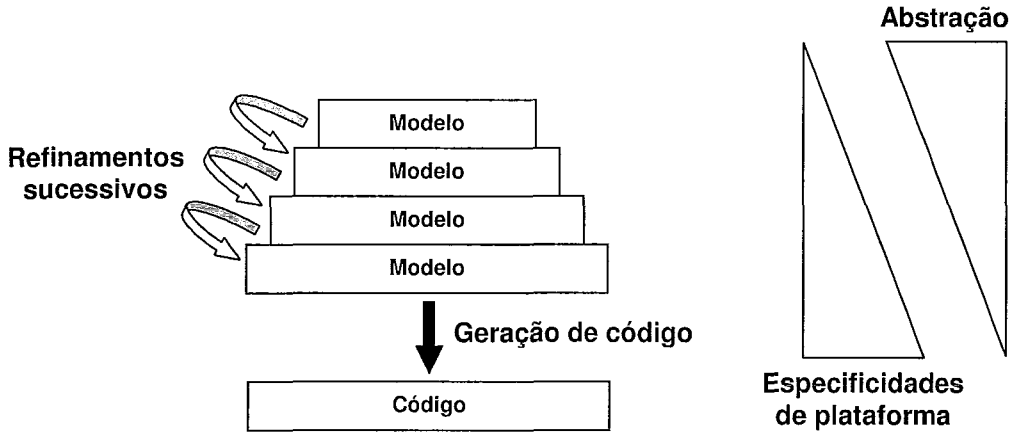

Figura 2.1: Criação de modelos e código no MDD. Adaptado de *FONDEMENT et al.* (2004)

Para facilitar a aplicação do MDD, o OMG (*Object Management Group*) criou o MDA – *Model-Driven Architecture* (Arquitetura Dirigida por Modelos) (OMG, 2003b). O MDA especifica um grupo de modelos que possuem finalidades específicas. Entre esses modelos, podem ser destacados o PIM – *Platform Independent Model* (Modelo Independente de Plataforma) e o PSM – *Platform Specific Model* (Modelo Específico de Plataforma). O PIM é uma visão da solução sem referência a qualquer plataforma tecnológica. O PSM é uma visão do software que considera os aspectos tecnológicos de

uma plataforma particular. O PSM é usado normalmente para a geração do código fonte do aplicativo. Quando é necessário gerar o software para outra plataforma tecnológica, o mesmo PIM é usado para a geração do PSM da nova plataforma.

Em projetos de desenvolvimento de software de grande escala, profissionais com papéis específicos e localizados em lugares diferentes podem modificar os modelos do software independentemente. Por exemplo, um analista localizado em São Paulo pode fazer modificações no modelo de alto nível (PIM) em função de novos requisitos. Durante este período, projetistas no Rio de Janeiro podem fazer modificações no modelo de baixo nível (PSM) para completá-lo e, assim, permitir a geração automática do código fonte. Contudo, uma vez que esses modelos representam diferentes visões do mesmo software, é possível que a evolução independente de cada modelo torne-os inconsistentes um em relação ao outro. No caso do MDA, essas inconsistências podem gerar dificuldades para a geração de PSMs para outras plataformas tecnológicas. Por exemplo, projetistas podem fazer modificações no PSM que estão no nível de abstração do PIM. Nesse caso, quando o PIM for usado para a geração do PSM de outra plataforma tecnológica, o programa da nova plataforma não terá as mesmas funcionalidades que o programa da plataforma anterior, o que indica que não representam mais o mesmo software. O problema torna-se maior quando o projeto está focado em várias plataformas tecnológicas. Portanto, inconsistências entre os modelos do mesmo software não podem ser permitidas.

Para que os modelos de software possam evoluir consistentemente ao longo do tempo, é necessária uma abordagem que possibilite a transformação, sincronização e versionamento automático dos modelos.

Este capítulo apresenta a teoria necessária para a definição desta abordagem de controle de evolução de modelos, a ser aplicada ao MDD utilizando o *framework* MDA. A Seção 2.2 apresenta os conceitos da Arquitetura Dirigida por Modelos e sua aplicação no Desenvolvimento Dirigido por Modelos. A Seção 2.3 discute os aspectos relacionados com a consistência entre modelos. A Seção 2.4 trata da consistência intra-modelo. A Seção 2.5 apresenta os principais conceitos sobre Gerência de Configuração, Controle de Versão, a aplicação desses recursos no desenvolvimento de software, e o controle de versão de modelos. Finalmente, na Seção 2.6, são apresentadas as considerações finais sobre o capítulo.

2.2 Arquitetura Dirigida por Modelos (MDA)

A Arquitetura Dirigida por Modelos é ao mesmo tempo um conjunto de padrões e um *framework* para o MDD. Uma das finalidades do MDA é a separação da especificação das funcionalidades do software de sua implementação em uma plataforma tecnológica específica. Desse modo, os desenvolvedores podem concentrar maior esforço na especificação das funcionalidades e na criação de uma solução genérica, sem a preocupação com as particularidades tecnológicas das plataformas de software para as quais o software será desenvolvido. A partir da solução genérica, é possível gerar o sistema para qualquer plataforma de software dentro de prazos e custos menores (FLATER, 2002). Portanto, o MDA possibilita o rápido desenvolvimento de sistemas independentemente de plataformas de software.

Para agilizar o processo de desenvolvimento, o MDA considera a geração automática de novos modelos a partir de outros através de transformações. Um processo de transformação utiliza as características do modelo de origem mais as regras definidas por uma **especificação de transformações** para gerar o novo modelo. A última transformação realizada é a geração do código fonte para uma plataforma tecnológica específica. Os padrões, conceitos e aplicação do MDA são detalhados nas próximas seções.

2.2.1 Modelos MDA

Um modelo é uma representação que tem como objetivo facilitar a compreensão de conceitos físicos a abstratos de um sistema existente ou a ser construído (BOOCH *et al.*, 2005). No contexto do desenvolvimento de software, um modelo pode ser usado para facilitar o entendimento do problema a ser resolvido ou da solução a ser implementada. Além disso, modelos permitem que as pessoas envolvidas no desenvolvimento possam comunicar-se melhor. Portanto, modelos são elaborados com o objetivo de permitir que os desenvolvedores possam lidar de maneira mais eficaz com o tamanho e a complexidade inerentes ao tipo de software que deve ser criado, de modo a atender às necessidades das organizações e usuários contemporâneos. Contudo, no MDA, os modelos também passam a ser usados como artefatos para a geração automática do software para uma plataforma tecnológica (BROWN, 2004).

Os quatro modelos definidos no MDA são: Modelo Independente de Computação (**CIM** – *Computation Independent Model*), Modelo de Plataforma (**PM** -

Platform Model), PIM e PSM, estes últimos definidos na seção anterior. Esses modelos encontram-se em diferentes níveis de abstração, sendo o CIM o modelo mais abstrato e o PSM o menos abstrato. O PIM está em um nível de abstração intermediário, entre o CIM e o PSM.

O **CIM**[3] é usado para representar conceitos e requisitos do negócio, como, por exemplo, o vocabulário usado e as regras do negócio (LINEHAN, 2008). O CIM é uma visão do ambiente no qual o sistema a ser desenvolvido será implantado.

O **PIM**, conforme já descrito, é um modelo que representa uma solução computacional genérica e independente de qualquer plataforma. Esse modelo pode ser usado para gerar o PSM para qualquer plataforma tecnológica que tenha compatibilidade com os padrões de arquitetura usados para a construção do PIM.

O **Modelo de Plataforma** (PM – *Platform Model*) representa os conceitos técnicos de uma plataforma tecnológica. Uma plataforma é um conjunto de subsistemas e recursos tecnológicos que fornecem serviços através de padrões de uso e interfaces bem definidas. Um sistema implementado para uma plataforma específica executa as suas funções a partir dos serviços oferecidos pela plataforma, mas sem ter conhecimento de como os serviços são implementados ou executados. Exemplos de plataformas de software são o CORBA (OMG, 2002), o JEE (SUN, 2007) e o .NET (MICROSOFT, 2007). Portanto, o Modelo de Plataforma define os serviços e padrões de uma plataforma e como esses devem ser aplicados ao PIM, de modo que o PSM gerado tenha como ser usado para a geração do código para a plataforma informada.

O **PSM**, descrito na seção anterior, é um modelo da solução computacional criado a partir da combinação da solução computacional independente de plataforma (PIM), com os recursos de uma plataforma tecnológica específica. Trata-se de uma junção do PIM com o PM.

2.2.2 Linguagem para Representação de Modelos

Os modelos de software precisam ser representados de alguma maneira. A linguagem padrão do MDA para a criação de modelos é a UML – *Unified Model Language* (Linguagem de Modelagem Unificada) (BOOCH *et al.*, 2005). Além de permitir a modelagem do software utilizando diferentes perspectivas, a UML permite

[3] Este trabalho está direcionado apenas para modelos computacionais. Por esse motivo, o CIM não está sendo considerado.

que seus elementos sejam estendidos a partir do uso de perfis (*Profiles*) (OMG, 2007). Um perfil permite a definição de estereótipos[4] que podem ser aplicados aos elementos UML. Esses estereótipos podem conter atributos que servem como valores etiquetados (*tagged values*), usados para manter dados adicionais dos elementos que constituem o modelo, como, por exemplo, informar se um elemento pode ser transformado ou não. Os perfis UML são essenciais para a transformação de modelos, conforme será explicado na seção 2.2.3.

2.2.3 Transformação de Modelos

A transformação de modelos é o processo de criar novos modelos (modelos-alvo) do mesmo software a partir de um ou mais modelos existentes (modelos-fonte). De acordo com a especificação do MDA (OMG, 2003b), os métodos para a realização de transformação são: (1) manual; (2) baseada em perfil; (3) baseada em padrões e marcações; e (4) automática. A **transformação manual** é caracterizada pela criação manual do PSM a partir do PIM. Na **transformação baseada em perfil**, um especialista acrescenta marcações (estereótipos e valores etiquetados) no PIM, usando como referência um perfil UML independente de plataforma. Em seguida, o modelo é transformado em um PSM e recebe marcações baseadas em um perfil de uma plataforma específica. **Transformações baseadas em padrões e marcações** são caracterizadas pelo uso de padrões para definir mapeamentos entre os elementos dos modelos de origem e de destino e pelo uso de marcações que permitem identificar o padrão a ser usado para criar elementos do(s) modelo(s)-alvo a partir de elementos do(s) modelo(s)-fonte. A **transformação automática** é a geração de código fonte a partir do PIM, sem a criação do PSM. Para isso, o PIM deve ser completo, contendo não apenas a estrutura, mas também o comportamento do sistema.

Apesar de uma transformação poder ser realizada manualmente, o objetivo é a execução automática ou semi-automática dessa operação. A automatização é essencial para que o MDD apresente vantagens reais em relação às abordagens tradicionais de desenvolvimento de software (BILLIG *et al.*, 2004), principalmente no que diz respeito ao prazo de entrega. Sem esse recurso, a diferença do MDD para as demais abordagens

[4] Um estereótipo permite classificar ou fornecer alguma informação adicional sobre um elemento, como, por exemplo, informar que uma classe pertence à camada de interface (MELLOR *et al.*, 2004).

é a existência de regras bem definidas para a criação de modelos-alvo a partir de modelos-fonte.

Para que a execução automática seja realizada, é necessário implementar um mecanismo para gerar modelos-alvo a partir de modelos-fonte. Apesar de ser possível criar algoritmos para fazer a transformação de forma direta, a melhor abordagem é a implementação de rotinas que utilizam mapas (ou regras) de transformação. A vantagem de usar mapas de transformação é permitir que os desenvolvedores possam definir novas transformações sem que o programa tenha que ser compilado novamente.

Uma **especificação de transformação** fornece os mapas de transformação que determinam como elementos do modelo-alvo devem ser criados a partir de elementos do modelo-fonte. Esses mapas, normalmente, são baseados em marcas específicas, como estereótipos e valores etiquetados. Por exemplo, um mapa de transformação pode determinar que classes PIM que possuem o estereótipo `<<Entidade>>` devem gerar classes EJB (*Entreprise Java Bean*) para a plataforma JEE (SUN, 2008). As especificações de transformação PIM-PSM são criadas com base no Modelo da Plataforma.

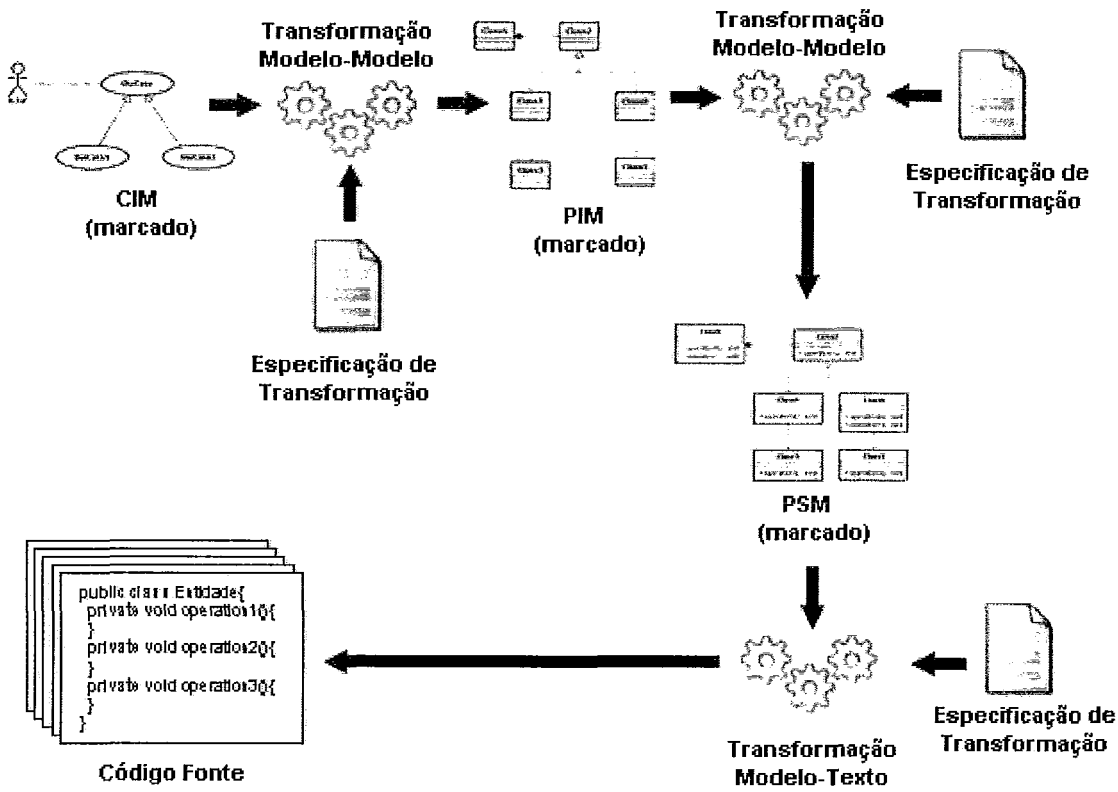

Figura 2.2: Processo de transformação do MDA

O processo de transformação usando especificações de transformação de modelos é apresentado na Figura 2.2. Para a geração de código para uma plataforma de software específica, uma transformação modelo-modelo[5] utiliza o CIM e uma especificação de transformação para gerar o PIM. Em seguida, outra transformação modelo-modelo usa o PIM e outra especificação de transformação voltada para uma plataforma específica para criar o PSM. Finalmente, o PSM é usado por uma transformação modelo-texto[6] para gerar o código fonte para a plataforma tecnológica desejada (OMG, 2003b).

No contexto do MDA, as duas principais abordagens para a transformação de modelos são a traducionista e a elaboracionista (HAYOOD, 2004). A **abordagem traducionista**, também conhecida como UML executável (MELLOR *et al.*, 2002), tem como finalidade gerar o programa sem que o usuário tenha que fazer qualquer modificação no PSM ou no código fonte. Assim, o código pode ser imediatamente compilado para a geração do programa. Contudo, para que o código fonte seja completo, é necessário que o PIM represente toda a estrutura e o comportamento do software. A estrutura pode ser especificada a partir de modelos de classes e componentes. O comportamento é especificado a partir de linguagens de ações semânticas, como a *Action Specification Language* (*ASL*) (KENNEDY-CARTER, 2006) e a *Object Action Language* (*OAL*) (ACCELERATED-TECHNOLOGY, 2006). O processo de transformação da abordagem traducionista é apresentado na Figura 2.3. Exemplos de ferramentas que usam a abordagem traducionista são o xUML (KENNEDY-CARTER, 2006) e o *Nucleus BridgePoint* (ACCELERATED-TECHNOLOGY, 2006).

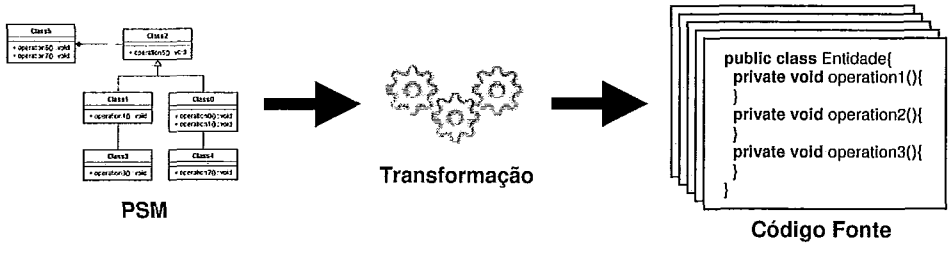

Figura 2.3: Transformação da abordagem traducionista

[5] Recebe modelos como entrada e gera modelos como saída.

[6] Recebe modelos como entrada e gera texto, como, por exemplo, o código-fonte.

A **abordagem elaboracionista** é caracterizada pela existência do PSM como modelo intermediário entre o PIM e o código fonte, conforme apresentado na Figura 2.4. Além disso, o PIM, normalmente, não apresenta a especificação do comportamento da aplicação. O comportamento é inserido a partir do refinamento do PSM e do código fonte. Exemplos de abordagens elaboracionistas são o *ArcStyler* (INTERACTIVE-OBJECTS, 2006) e o *OptimalJ* (COMPUWARE, 2007).

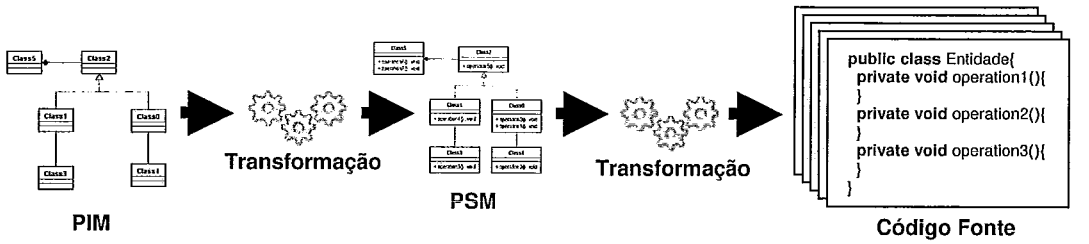

Figura 2.4: Transformação da abordagem elaboracionista

As transformações podem ser unidirecionais ou bidirecionais (CZARNECKI *et al.*, 2003). **Transformações unidirecionais** são caracterizadas pela geração dos modelos-alvo a partir dos modelos-fonte. Por outro lado, **transformações bidirecionais** podem ser realizadas em ambas as direções, o que significa que um modelo pode ser criado a partir do outro. Neste caso, esses modelos são definidos como **modelos da esquerda** e **modelos da direita**. Desse modo, quando a direção da transformação é da esquerda para a direita, o modelo da direita é gerado a partir do modelo da esquerda, e vice-versa. Normalmente, a transformação esquerda-direita determina uma engenharia adiante (*forward engineering*), enquanto a transformação direita-esquerda determina uma engenharia reversa.

Transformações bidirecionais podem ser especificadas a partir de dois mapas de transformação unidirecionais separados, um para cada direção da transformação, ou a partir de um mapa de transformação que contempla a transformação para ambas as direções.

2.2.4 Registro de Transformação

Durante a geração do modelo-alvo, a transformação pode gerar **registros de transformação** (RT) (OMG, 2003b). Um RT (Figura 2.5), ou ligação de rastreabilidade, relaciona um elemento do modelo-fonte com um elemento do modelo-alvo gerado pela transformação, além da regra (ou mapa) de transformação que foi

usada. O RT permite a identificação de um elemento a partir de outro, além da regra de transformação usada. O RT é útil para a análise de impacto de modificações (como as mudanças de um modelo podem afetar outros modelos) e sincronização de modelos (identificação de elementos de um modelo que devem ser atualizados em função de alterações realizadas em elementos de outro modelo) (CZARNECKI *et al.*, 2003). Portanto, RTs são essenciais para controlar e evolução de modelos inter-relacionados.

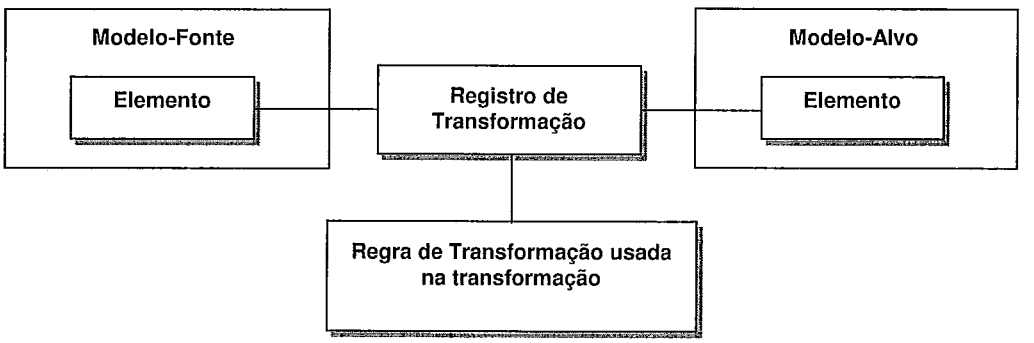

Figura 2.5: Registro de Transformação

Um aspecto importante em relação às ligações de rastreabilidade é onde manter as respectivas informações. Segundo Sendall *et al* (2004), duas abordagens possíveis são: colocar as informações dos rastros nos próprios modelos ou no código (a partir de comentários), ou armazenar as informações externamente. A primeira abordagem é muito comum quando as informações de rastreabilidade são usadas para a sincronização de classes, implementadas em uma linguagem de programação, com as respectivas representações em UML. Porém, os comentários podem ser apagados e o código pode se tornar mais difícil de entender. A segunda abordagem resolve essas limitações. De qualquer modo, um problema que pode surgir nas duas abordagens é diferentes ferramentas terem estruturas de rastreabilidade proprietárias e, desse modo, a interoperabilidade entre as ferramentas pode não ser possível. Para resolver este problema, a OMG está propondo o QVT (*Query View Transformation*) (OMG, 2008), um conjunto de notações para a especificação de consultas, visualização e transformações. Antes mesmo de sua versão final, o QVT foi indicado por SENDALL *et al* (2004) como recurso para padronizar as informações de rastreabilidade geradas a partir da transformação de modelos.

As abordagens para persistência dos rastros podem ser classificadas em explícitas e implícitas (IVKOVIC *et al.*, 2004). **Abordagens explícitas** mantém as

ligações de rastreabilidade separadamente, como, por exemplo, em uma tabela. Essa estrutura precisa ser atualizada sempre que o modelo-fonte ou o modelo-alvo for modificado. Em **abordagens implícitas**, as dependências são definidas entre os meta-modelos, tornando implícito o relacionamento entre os modelos. A segunda abordagem é mais flexível e adaptável, além de facilitar a manutenção.

2.2.5 Níveis de Abstração de Modelos

Durante o desenvolvimento de software realizado a partir do MDD, diferentes modelos podem ser gerados. Esses modelos podem estar no mesmo nível de abstração ou em níveis de abstração diferentes. No contexto do MDA, CIM, PIM e PSM estão em níveis diferentes de abstração. Desse modo, as transformações CIM-PIM, PIM-PSM e PSM-Código são **transformações verticais** (SENDALL *et al.*, 2003). Contudo, é possível a aplicação de **transformações horizontais**, de modo que modelos no mesmo nível de abstração sejam gerados (SENDALL *et al.*, 2003). Assim, é possível realizar transformações PIM-PIM e PSM-PSM. Transformações horizontais são usadas para o refinamento de modelos. Por exemplo, uma transformação horizontal PIM-PIM pode ser usada para gerar um modelo-alvo a partir da aplicação de padrões de projeto ao modelo-fonte.

Dependendo do número de transformações necessárias para se conseguir um modelo suficientemente completo para a geração do código fonte, um modelo-alvo pode ser um modelo-fonte para uma outra transformação, que irá gerar outro modelo-alvo. Desse modo, é possível existir uma série de modelos-fonte e modelos-alvo inter-relacionados. Além disso, esta situação pode ocorrer com modelos no mesmo nível de abstração ou níveis de abstração diferentes. Assim, no contexto do MDA, um PSM pode ser um PIM para uma transformação vertical que irá gerar um novo PSM em outro nível de abstração (OMG, 2003b). Por exemplo, um PIM pode ser usado para a geração de um PSM para uma plataforma de componentes. Esse modelo pode ser usado como um PIM para criar um PSM para a plataforma JEE (SUN, 2007). Os possíveis relacionamentos que podem existir entre modelos de um projeto MDD podem ser observados na Figura 2.6. As linhas destacadas demonstram um exemplo de modelos inter-relacionados.

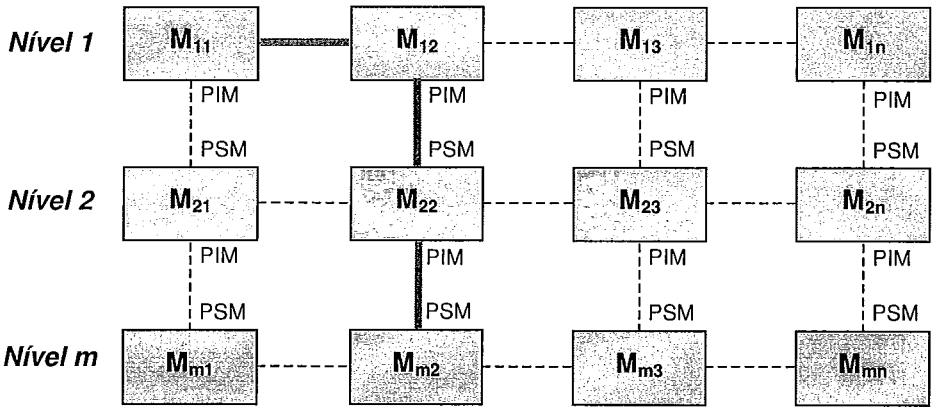

Figura 2.6: Modelos Inter-Relacionados

2.2.6 O Metadata Object Facility (MOF)

O *Metadata Object Facility (MOF)* é o *framework* adotado pela OMG para a criação de aplicativos orientados a metadados e metamodelos e para possibilitar a interoperabilidade entre sistemas (OMG, 2006). O MOF é usado para a especificação de meta-modelos de linguagens de modelagem, como a UML e o CWM (*Commom Warehouse Metamodel*) (OMG, 2003a). Além disso, é um recurso essencial para definir transformações de modelos que possuem metamodelos diferentes (MAIA, 2006).

A especificação 1.4 do MOF (OMG, 2005a) apresentava uma organização em camadas com quatro níveis de abstração, conforme a Tabela 2.1. Por ser usado para a especificação de metamodelos, o MOF era considerado como um *meta-meta-modelo*, sendo o nível mais alto de abstração da arquitetura de metamodelos da OMG. Contudo, esta organização não foi utilizada na especificação 2.0 do MOF (OMG, 2006) por ter sido considerada muito restritiva. Portanto, o MOF 2.0 permite qualquer número de camadas, sendo necessário pelo menos duas: metamodelo e modelo. Assim, elementos do metamodelo podem ser usados para representar elementos do modelo. Todavia, é importante notar que apesar de não ser mais oficialmente parte da especificação do MOF, a organização em quatro camadas ainda é usada como referência na especificação da infraestrutura da UML 2 (OMG, 2007).

2.2.7 Interoperabilidade de Modelos

Diferentes cenários podem fazer com que um modelo tenha que ser manipulado por ferramentas diferentes. Alguns exemplos são: migração de uma ferramenta CASE

para outra, cópia de um modelo de um aplicativo para outro através da área de transferência do sistema operacional, uso de ferramentas com finalidades específicas para criar e manter modelos e transferência e recuperação de modelos em repositório (ALANEN *et al.*, 2005).

Tabela 2.1: Níveis de metadados da OMG.

Nível		Descrição
M3	Meta-Metamodelo	Modelo MOF (possui um conjunto de elementos usados para especificar os elementos de um metamodelo). Exemplo: uma classe MOF é usada para especificar classes, estados e componentes da UML, enquanto um atributo MOF é usado para especificar as propriedades de cada um desses elementos, como determinar que uma classe UML possui a propriedade chamada de atributo.
M2	Metamodelo	Um metamodelo é definido a partir do meta-metamodelo MOF e é usado para definir modelos. Exemplo: Uma classe UML é usada para especificar uma classe de objetos referente a um modelo, como a classe Cliente.
M1	Modelo, Metadado	Representam objetos e comportamentos. É o nível em que os modeladores trabalham, abstraindo o que é necessário do mundo real e criando os modelos necessários. Exemplo: um modelo de classes, com uma classe com o nome Cliente e os atributos nome, sexo e cidade onde mora.
M0	Instância de Modelo	São objetos e dados do mundo real. Exemplo: um cliente chamado Roberto, com idade de 34 anos, de sexo masculino e morador da cidade do Rio de Janeiro.

Para possibilitar a interoperabilidade de modelos entre diferentes ferramentas, a OMG criou o XMI (*XML MedaData Interchange*) (OMG, 2005b). Esta especificação possui um conjunto de regras para o mapeamento de metamodelos baseados no MOF (*Meta Object Facility*) (nível M2) e dos respectivos modelos (nível M1) em documentos XML. Desse modo, quaisquer modelos cujos metamodelos foram especificados em MOF podem ser importados e exportados através de arquivos XMI, independentemente da forma como são representados internamente por cada ferramenta de modelagem.

Apesar da importância do XMI para o compartilhamento de modelos, ALANEN e PORRES (2005) relatam problemas relacionados às versões do XMI e metamodelos diferentes, identificação de metamodelos e identificação de elementos. A existência de versões diferentes do XMI e de metamodelos, como o da UML, por exemplo, faz com que os modelos sejam representados de maneiras diferentes, tornando necessário que os mecanismos de importação de modelos sejam baseados em uma única combinação de versões, ou que sejam capazes de converter versões mais antigas para versões mais

novas. A identificação de metamodelos é necessária para que as ferramentas possam processar os modelos de forma correta. Contudo, o XMI não possui uma forma confiável de informar o metamodelo usado para a geração do documento. O processo de identificação de elementos atribui um identificador único para cada elemento, normalmente um UUID[7] (*Universally Unique Identifier*). O identificador, depois de atribuído a um elemento, não deveria ser modificado. Contudo, muitas ferramentas não preservam os identificadores, dificultando tarefas como a comparação e junção de modelos, criação e manutenção de rastros e sincronização de modelos. Esses problemas não inviabilizam o uso do XMI, mas devem ser levados em consideração no momento da especificação ser usada.

2.2.8 Aplicação do MDA no Desenvolvimento Dirigido por Modelos

A aplicação do MDA no Desenvolvimento Dirigido por Modelos pode ser dividida em duas etapas: elaboração dos mapeamentos de transformação e desenvolvimento (Figura 2.7). Na etapa de elaboração de mapeamentos, o projetista de transformações cria os mapeamentos necessários para gerar o modelo-alvo. No caso de transformações para a criação do PSM para uma plataforma particular, o Modelo de Plataforma é usado como base para a criação dos mapeamentos. Além dos mapeamentos, pode ser necessário definir um perfil contendo os estereótipos e valores etiquetados para a marcação dos elementos do modelo-fonte e do modelo-alvo (a transformação pode marcar automaticamente o modelo alvo com base nas regras de mapeamento).

Na etapa de desenvolvimento, um engenheiro de software pode criar o modelo-fonte usando uma ferramenta de modelagem. Em seguida, o modelo recebe as marcações necessárias. Essas marcações podem ser estereótipos ou valores etiquetados definidos em um perfil UML. Finalmente, o engenheiro de software aciona o mecanismo de transformação, informando o modelo-fonte marcado e o mapeamento de transformação. O resultado é a geração do modelo-alvo e dos registros de transformação. O modelo-alvo pode ser modificado e ser usado como modelo-fonte para a geração de outro modelo. Por exemplo, um PIM pode ser usado para a geração de um PSM, e este pode ser usado para a geração do código fonte. Apesar desse cenário estar

[7] Um UUID é um *string* de 128 bits de valores hexadecimais, sendo único em todo o mundo (IETF, 2005).

focado na engenharia adiante (*forward engineering*), é possível realizar engenharia reversa através de transformações que geram o modelo-fonte a partir do modelo-alvo, como, por exemplo, a criação do PIM a partir de um PSM.

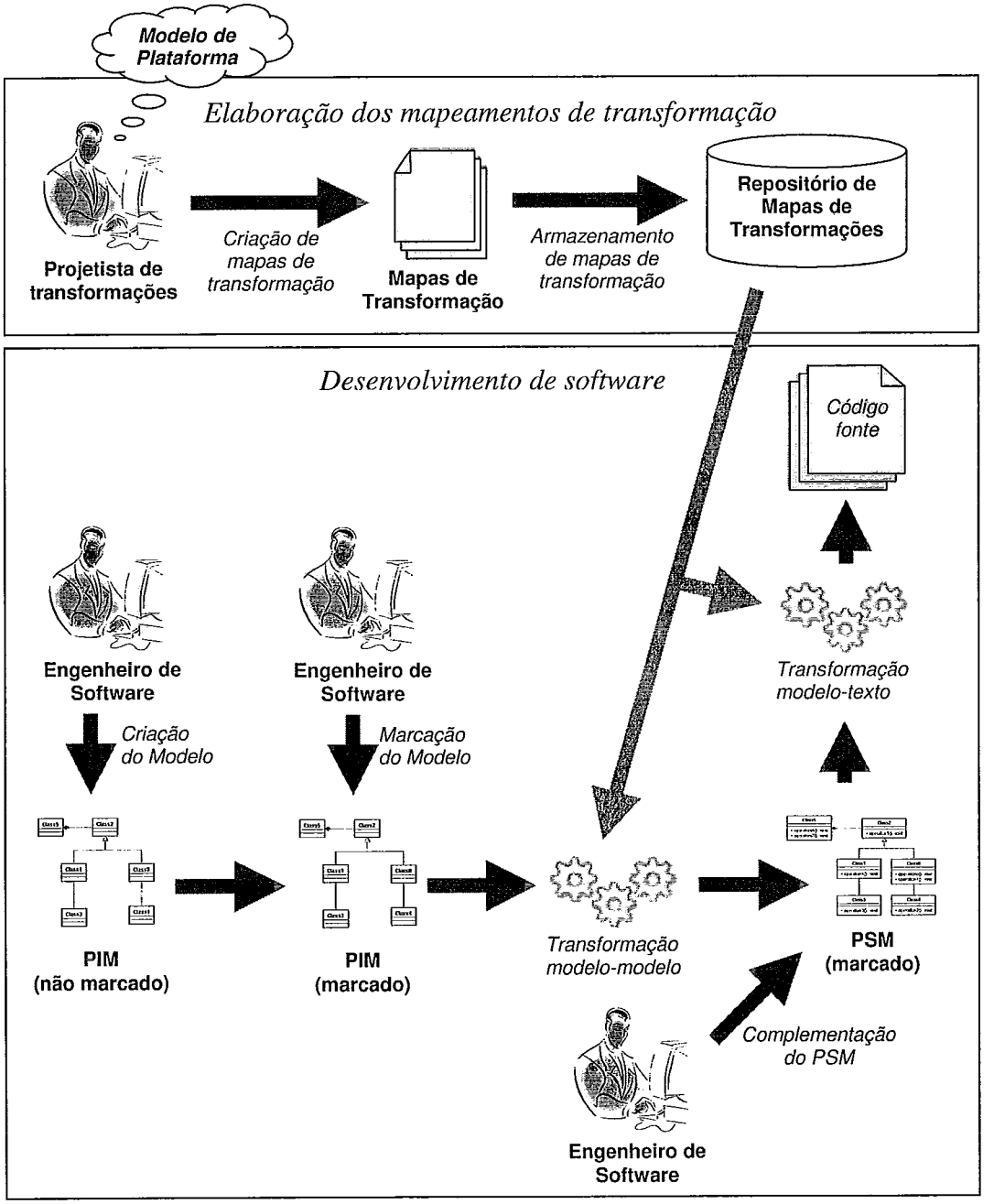

Figura 2.7. Aplicação do MDA no Desenvolvimento Orientado a Modelos

2.3 Consistência entre Modelos

Modelos de software inter-relacionados possuem elementos que fazem referência a aspectos comuns do mesmo sistema (SPANOUDAKIS *et al.*, 2001). Ao

longo do ciclo de desenvolvimento e de manutenção do software, esses modelos podem ser modificados independentemente, principalmente quando estão fisicamente separados ou quando pessoas com diferentes papéis precisam fazer alterações em função de necessidades específicas. Desse modo, os modelos inter-relacionados podem evoluir independentemente e tornarem-se inconsistentes, o que significa que podem existir contradições em relação aos aspectos comuns do sistema que são representados por esses modelos. Contudo, como representam o mesmo software, inconsistências não podem ser permitidas. Para que modelos possam evoluir de maneira consistente, é necessário **sincronizá-los**. Essa tarefa pode ser definida como o processo de manter a equivalência de modelos diferentes quando um deles é modificado (IVKOVIC *et al.*, 2004) (consistência inter-modelos). A sincronização deve manter consistentes modelos no mesmo nível de abstração (consistência horizontal) e em níveis de abstração diferentes (consistência vertical) (ENGELS *et al.*, 2002).

No contexto do MDD, modelos normalmente são usados como referência para a criação de outros modelos através de transformações. Portanto, alguns modelos são usados como entrada (modelos-fonte) para o mecanismo de transformação gerar modelos de saída (modelos-alvo). Depois da transformação, alterações podem ser realizadas em qualquer um dos modelos. Por esse motivo, a sincronização deve ser realizada em **duas direções**, ou seja, dos modelos da esquerda para os modelos da direita, e vice-versa. Desse modo, quando um PIM é modificado, o PSM de cada plataforma deve ser sincronizado; quando o PSM de uma plataforma é alterado, o PIM e o PSM das outras plataformas devem ser atualizados, se for o caso.

A necessidade de se manter os modelos inter-relacionados do MDD sincronizados caracteriza um trabalho adicional para o desenvolvimento de software orientado a modelos, sendo considerado como o principal problema indicado pelos desenvolvedores (FORWARD *et al.*, 2008). Para evitar esse trabalho, a tarefa deve ser automatizada. A capacidade de se manter diferentes artefatos de software consistentes através de sincronização bidirecional e automática é conhecida como **engenharia *roundtrip*** (ida e volta) (SENDALL *et al.*, 2004). De certa forma, é a combinação da engenharia adiante com a engenharia reversa. Porém, enquanto a engenharia adiante e a engenharia reversa geram sempre novos artefatos, normalmente substituindo versões anteriores, a engenharia *roundtrip* procura preservar as particularidades dos artefatos antes da sincronização. Portanto, além de geração, outra finalidade da engenharia *roundtrip* é a reconciliação de modelos.

Para que a engenharia *roundtrip* (sincronização bidirecional e automática) tenha como ser realizada, existem alguns aspectos a serem levados em consideração, como as propriedades da sincronização, a localização e quantidade dos elementos a serem atualizados e quando as modificações de um modelo devem ser propagadas para outros modelos. Esses aspectos são discutidos nas seções a seguir.

2.3.1 Propriedades da Sincronização

XIONG et al. (2007), usando como referência as propriedades das transformações de árvores bidirecionais (*bidirection tree transformations*), propuseram as seguintes propriedades para a sincronização de modelos: estabilidade, preservação, propagação e capacidade de compor (*composability*). As descrições dessas propriedades são apresentadas a seguir:

- *Estabilidade*: A sincronização não deve modificar os modelos, caso nenhum tenha sido modificado.
- *Preservação*: A sincronização deve preservar as modificações de todos os modelos.
- *Propagação*: A sincronização deve garantir a propagação correta das modificações para todos os modelos.
- *Capacidade de compor*: Existindo duas seqüências de operações de sincronização, uma sincronização executada duas vezes, com as duas seqüências, deve gerar o mesmo resultado que executar a sincronização apenas uma vez, com uma seqüência de operações composta das duas seqüências anteriores.

Caso a sincronização seja aplicada aos modelos que não foram modificados, a propriedade *estabilidade* garante que os modelos permanecerão os mesmos após o término de sua execução. A *preservação* possibilita que as modificações realizadas em elementos de um modelo que não estão relacionados com outros modelos sejam preservadas. Por outro lado, a *propagação* determina que modificações realizadas em um modelo que tenha relação com outros sejam propagadas para esses modelos. Desse modo, a sincronização deve sempre ser realizada em duas direções, tornando viável a engenharia *roundtrip*. Finalmente, a *capacidade de combinar* tem como finalidade permitir que a sincronização tenha como ser realizada a qualquer momento.

2.3.2 Localização de Elementos

A localização tem relação direta com as regras de transformação que foram usadas para a geração do modelo-alvo. Por exemplo, uma classe do PIM com o nome `Cliente` e com o estereótipo `<<Entidade>>` pode ser usada pelo mecanismo de transformação para a geração da classe PSM `ClienteBean`, para a plataforma JEE. Desse modo, existe uma **ligação de rastreabilidade** entre essas duas classes. Se a classe `Cliente` for modificada, a classe `ClienteBean` deve ser atualizada; se a classe `ClienteBean` receber alguma alteração que deve ser representada na classe `Cliente`, é necessário atualizá-la. Portanto, além do conhecimento dos elementos que possuem relacionamento de rastros, é necessário saber quando e que tipo de atualização deve ser realizada. Esse tipo de informação pode ser obtido a partir da regra (ou mapa) de transformação que foi usada.

2.3.3 Quantidade de Elementos

A quantidade de elementos a serem atualizados pode variar em função da transformação realizada. Transformações podem gerar vários elementos para o modelo-alvo a partir de um único elemento do modelo-fonte. Do mesmo modo, um único elemento do modelo-alvo pode ser resultado da transformação de vários elementos do modelo-fonte. Ou ainda, vários elementos do modelo-fonte podem ser usados para gerar vários elementos do modelo-alvo. Desse modo, algumas transformações podem gerar relações de rastro com multiplicidade de muitos-para-muitos entre os elementos de modelos diferentes. Portanto, é possível que vários elementos tenham que ser atualizados em função da alteração de um único elemento de outro modelo.

2.3.4 Propagação de Modificações

Uma das propriedades da sincronização apresentada na seção 2.3.1 é a propagação de modificações, que garante a atualização dos modelos inter-relacionados sempre que um modelo é modificado. A propagação depende da inter-relação existente entre os elementos de cada modelo. A inter-relação existe quando um elemento é gerado a partir do elemento de outro modelo, através de uma transformação. Nesse caso, sempre que um desses elementos for modificado, os elementos inter-relacionados devem ser atualizados. Do mesmo modo, a inclusão e remoção de elementos em um modelo

devem implicar nas mesmas operações nos elementos inter-relacionados de outros modelos.

Por outro lado, a propagação de modificações não deve ocorrer quando as modificações realizadas em um modelo não possuem relação com outros modelos. Isso ocorre quando não existem regras de transformação relacionadas com as alterações realizadas. Por exemplo, classes de um PSM podem conter métodos privados que foram inseridos pelo desenvolvedor, de modo a tornar o modelo mais completo para a geração de código fonte. Nesse caso, essas modificações não devem ser propagadas para o PIM.

Considerando a possibilidade de existir vários modelos inter-relacionados, é importante notar que as modificações realizadas em um modelo podem não ter relação com todos os outros modelos do projeto. Por exemplo, alterações em um PSM_A podem não ser propagadas para o PIM, mas podem ser propagadas para um PSM_B em outro nível de abstração, para o qual o PSM_A atua como PIM (Figura 2.8).

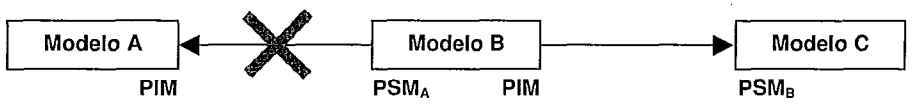

Figura 2.8. Propagação de alterações entre modelos diferentes

2.4 Consistência Intra-Modelo

Além da consistência entre modelos descrita na seção anterior, a evolução consistente de modelos depende da **consistência intra-modelo**, que é decorrente da relação que pode existir entre os elementos de um mesmo modelo. Por exemplo, para cada atributo da uma classe JEE que representa uma entidade[8], é recomendado o uso dos métodos *get* e *set*[9]. Assim, se uma classe *Cliente* possui um atributo *nome*, segundo a recomendação, a classe também deve ter os métodos *getNome* e *setNome*. Portanto, esses elementos estão relacionados. Nesse caso, para que o modelo permaneça consistente, a modificação ou a exclusão de um desses elementos implica na mesma operação nos outros elementos.

[8] Entidade é uma representação de um tipo de objeto que existe no domínio do problema em questão. Por exemplo, cliente, venda e veículo são entidades do domínio de venda de veículos.

[9] No J2EE,versão anterior do JEE, o uso de métodos *get* e *set* eram obrigatórios (SUN, 2007).

2.5 Gerência de Configuração e Controle de Versão

Programas de computador estão sujeitos a modificações em função de novas necessidades dos usuários, mudanças relacionadas aos requisitos existentes, aperfeiçoamentos e correção de defeitos. Essas mudanças podem ocorrer durante todo o ciclo de vida de software (BERSOFF *et al.*, 1980).

A **Gerência de Configuração de Software** (GCS) é a disciplina que tem como finalidade permitir o controle da evolução de sistemas de software durante o desenvolvimento e manutenção, contribuindo para a qualidade e entrega de produtos dentro do prazo (ESTUBLIER, 2000). A GCS define como uma organização constrói e libera software e, normalmente é utilizada quando o software ou o processo de desenvolvimento apresenta complexidade em função dos problemas a serem resolvidos, do tamanho da solução, das tecnologias usadas e do número e da localização das pessoas que participam do desenvolvimento ou manutenção, sendo um importante recurso para o Desenvolvimento Global de Software (*Global Software Development*) (SANGWAN *et al.*, 2007).

A GCS possui os seguintes grupos de atividades (IEEE, 2004): (1) planejamento de GCS; (2) identificação de configuração; (3) controle de configuração (modificações); (4) construção e liberação; (5) contabilização; e (6) avaliação e revisão (auditoria). Neste contexto, o **controle de versão** (CV) é uma atividade que fornece suporte às atividades de identificação de configuração, controle de configuração e contabilização a partir da preservação do histórico evolutivo de todos os artefatos que são produzidos durante o desenvolvimento ou manutenção, incluindo os artefatos que constituem o próprio programa. A partir do CV, é possível o desenvolvimento simultâneo de versões diferentes de um mesmo sistema, além da possibilidade de recuperação de versões anteriores, dando maior flexibilidade e segurança para os desenvolvedores. Exemplos de sistemas para o controle de versão são o CVS (CEDERQVIST, 2005) e o *Subversion* (COLLINS-SUSSMAN et al., 2004).

Os conceitos relacionados ao controle de versão são apresentados nas subseções a seguir.

2.5.1 Item de Configuração, Versão e Configuração de Software

Um **item de configuração** (IC) é qualquer elemento que pode evoluir ao longo do tempo em função de modificações (IEEE, 2004). Exemplos de ICs são: diretórios,

arquivos de código fonte, documentos e modelos. Uma **versão** caracteriza o estado de um item de configuração em um determinado momento no tempo (IEEE, 2004). Sempre que um item é modificado, uma nova versão é criada. Desse modo, o controle de evolução de um IC é caracterizado pela manutenção de um histórico de todas as versões do IC.

Um item pode ser composto de outros ICs. Neste caso, a criação de uma nova versão de um dos itens que o compõe implica na criação de uma nova versão para o IC composto. Por exemplo, a alteração do atributo de uma classe implica na criação de uma nova versão da respectiva classe.

Finalmente, a **configuração de software** é caracterizada pela combinação de versões específicas dos artefatos que o compõem, como, documentos, modelos, código fonte, etc.

2.5.2 Sistemas de Controle de Versão

Os sistemas de controle de versão (SCV) têm como finalidade controlar as modificações realizadas nos itens de configuração, gerando uma nova versão de um item sempre que é modificado e possibilitando que os artefatos permaneçam consistentes ao longo do tempo (consistência evolutiva) (ENGELS *et al.*, 2002). Além disso, SCVs que utilizam a política de controle de concorrência otimista (subseção 2.5.3) verificam a existência de conflitos e a junção de versões diferentes de itens de configuração. A **verificação de conflitos** averigua se as alterações realizadas por um desenvolvedor estão conflitando com alterações realizadas anteriormente por outro. A **junção** de ICs faz a união de uma versão existente no repositório com a cópia modificada pelo usuário. A junção é realizada quando não existem conflitos, caso contrário, o usuário precisa atualizar o seu espaço de trabalho e resolvê-los manualmente ou com o auxílio de alguma ferramenta.

Além das funcionalidades anteriores, dois outros recursos que, normalmente, são oferecidos por um SCV são: a criação das configurações do software que serão liberadas para os interessados (*releases*) e a possibilidade de criar ramos distintos a partir dos quais configurações específicas podem evoluir independentemente. Por exemplo, um ramo pode ser usado para a realização de correções enquanto outro pode ser usado para evoluir o software.

Os sistemas de controle de versão (SCV), usualmente, utilizam uma arquitetura cliente-servidor para manter os ICs. No lado servidor, existe um **repositório** onde o

SCV armazena o histórico de versões de todos os itens de configuração de cada projeto de software. Quando é necessário fazer alguma alteração, o desenvolvedor utiliza um programa cliente para solicitar ao SCV os artefatos a serem modificados (***check-out***). Esses artefatos são colocados no **espaço de trabalho** do usuário, onde podem ser alterados livremente, sem afetar as alterações dos outros desenvolvedores. Depois que as modificações estão prontas, o desenvolvedor utiliza a ferramenta cliente para enviá-las para o SCV (***check-in***), que irá fazer a verificação de conflitos, a junção e o versionamento dos ICs modificados. Após essas tarefas, o SCV armazena a nova versão do artefato no repositório, tornando-o disponível para *check-out*. Um esquema de um sistema de controle de versão é apresentado na Figura 2.9.

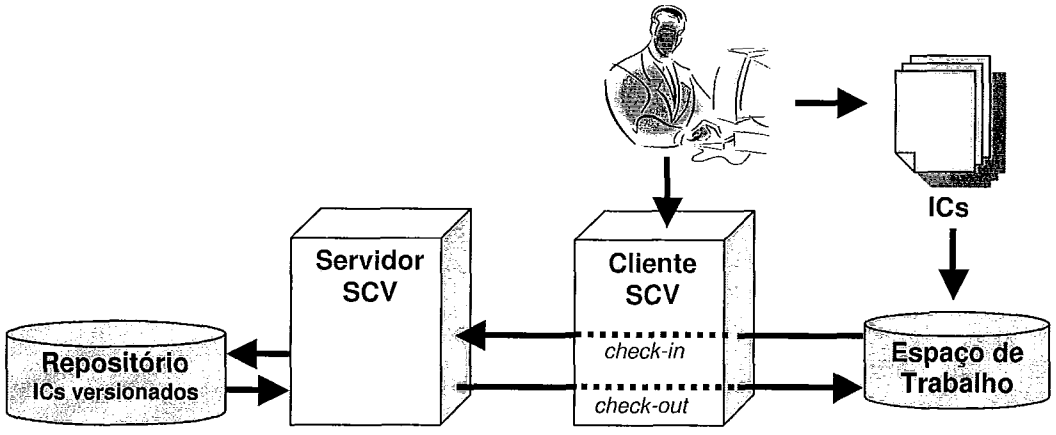

Figura 2.9. Sistema de Controle de Versão

2.5.3 Políticas para Controle de Concorrência

A possibilidade de mais de um desenvolvedor trabalhar em um mesmo artefato caracteriza o trabalho concorrente. Desse modo, alterações realizadas por um desenvolvedor podem entrar em conflito com as alterações realizadas por outro. Para resolver esse problema, pode ser adotada uma política de concorrência pessimista ou otimista. A **política pessimista** impede que os desenvolvedores tenham liberdade de modificar o mesmo artefato ao mesmo tempo. Desse modo, somente o desenvolvedor que fez o *check-out* do artefato pode modificá-lo. Nesse caso, os demais desenvolvedores precisam aguardar pela liberação do artefato para que possam fazer outras alterações (ESTUBLIER *et al.*, 2003). Essa política impede a existência de conflitos, evitando modificações concorrentes. Contudo, o tempo que o artefato permanece bloqueado pode atrasar o desenvolvimento ou a manutenção do software.

A **política otimista** assume que a possibilidade de conflitos é pequena. Desse modo, desenvolvedores diferentes podem modificar o mesmo artefato livremente e ao mesmo tempo. Quando ocorre algum conflito, o desenvolvedor precisa atualizar o seu espaço de trabalho e resolvê-lo antes de fazer um novo *check-in* (ESTUBLIER *et al.*, 2003). Quando não existem conflitos, é realizada a junção da versão do usuário com a versão existente no repositório. Para evitar a ocorrência de muitos conflitos, é necessário fazer a atualização freqüente do repositório.

2.5.4 Controle de Versão de Modelos

Aplicativos para controle de versão foram desenvolvidos inicialmente para controlar a evolução de arquivos em formato texto, particularmente código fonte. Neste cenário, cada arquivo é um IC e novas versões são criadas a partir do momento que linhas do texto são modificadas. Portanto, a linha é a unidade de comparação utilizada para a criação de uma nova versão (MURTA *et al.*, 2005). Exemplos de aplicações que permitem o versionamento de arquivos são o CVS (CEDERQVIST, 2005) e o Subversion (COLLINS-SUSSMAN *et al.*, 2004).

Apesar do controle de versão de arquivos ser amplamente utilizado para o controle da evolução do código fonte de aplicativos, essa abordagem não é adequada para modelos por dois motivos. Primeiro, os elementos que constituem um modelo, como, por exemplo, classes, associações, pacotes e componentes, também podem ser considerados como itens de configuração (MURTA *et al.*, 2005). Como na abordagem de controle de versão baseado em texto o menor grão de IC é o arquivo, seria necessário considerar todo o modelo como um único IC ou armazenar cada elemento em um arquivo separado. Contudo, ambas as soluções não são adequadas para o controle da evolução de modelos (KÖGEL, 2008).

O segundo motivo é o uso de linha como unidade de comparação para a realização de junção de modelos sem considerar a estrutura sintática do conteúdo do arquivo. A junção é realizada em função da mudança do conteúdo das linhas do arquivo, não importando a estrutura do modelo. Contudo, em vez de linhas, é necessário considerar mudanças realizadas sobre cada elemento que constitui o modelo e seus subitens, mesmo que o modelo esteja em um único arquivo (MURTA *et al.*, 2005).

Portanto, o versionamento de modelos requer SCVs que façam o controle de todos os elementos do modelo como itens de configuração, levando em consideração a estrutura sintática do modelo.

Apesar da importância de modelos no desenvolvimento de software, ainda não existem muitas ferramentas para controlar a evolução de modelos através de versionamento. Algumas ferramentas disponíveis são Odyssey-VCS (MURTA *et al.*, 2008), SCM *for* Sysiphus (KÖGEL, 2008) e o SMOVER (2008). Além do controle de versões, essas ferramentas são capazes de identificar conflitos, realizar a junção de modelos e permitir a atualização e recuperação de modelos a partir de *check-in* e *check-out*.

2.6 Considerações Finais

Sistemas de computador podem evoluir ao longo das etapas de desenvolvimento e manutenção. Essa evolução é decorrente da necessidade de fazer modificações devido à melhor compreensão, mudança e surgimento de requisitos, correção de defeitos e melhorias. Durante a execução de projetos segundo o paradigma MDD, diferentes modelos podem ser produzidos. Modelos gerados a partir de outros através de transformações estão fortemente inter-relacionados. Neste cenário, é necessário manter a consistência entre modelos no mesmo nível de abstração (consistência horizontal) e níveis de abstração diferentes (consistência vertical). Além disso, considerando a evolução dos modelos ao longo do tempo, é necessário manter a consistência entre as diferentes versões do mesmo modelo (consistência evolutiva).

Quando o desenvolvimento envolve várias pessoas trabalhando em equipe, existe a possibilidade de alguns profissionais precisarem trabalhar no mesmo modelo ou em modelos diferentes. Um sistema de controle de versão (SCV) pode ser usado para sincronizar a cópia modificada de um modelo com a versão atual através de uma operação de junção (*merge*). Quando conflitos são identificados, a versão original é preservada e o causador dos conflitos tem a responsabilidade de resolvê-los. Desse modo, o SCV é um recurso essencial para garantir a consistência evolutiva dos modelos.

Este capítulo apresentou uma visão geral sobre a teoria e os conceitos relacionados com o Desenvolvimento Dirigido por Modelos e a Gerência de Configuração de Software, destacando o controle de versão e a consistência de modelos. O Capítulo 3 apresenta as abordagens de desenvolvimento dirigido por modelos existentes na literatura.

Capítulo 3 – Abordagens MDD Relacionadas com a Evolução de Modelos

3.1 Introdução

No capítulo 2, foram discutidos os assuntos relacionados com o Desenvolvimento Dirigido por Modelos, consistência e controle de evolução de modelos ao longo do tempo e no espaço. Este capítulo apresenta as abordagens relacionadas de alguma maneira com a evolução de modelos dentro do contexto do MDD. A Seção 3.2 apresenta os critérios usados para a comparação e análise das abordagens. A Seção 3.3 descreve as abordagens e os critérios usados para a seleção. A análise comparativa das abordagens é apresentada na Seção 3.4.

3.2 Critérios para a Comparação das Abordagens

As abordagens são comparadas com base nos seguintes critérios, definidos a partir da revisão da literatura:

- *Utilização de linguagens e formatos padrões definidos pela OMG*: Indica se a abordagem utiliza os seguintes padrões da OMG: MOF, metamodelos baseados no MOF, como a UML, por exemplo, o QVT (*Query - View Transformation*) para a transformação de modelos, e uso do XMI para possibilitar a interoperabilidade dos modelos. A utilização de padrões aumenta o potencial de reutilização e de aprendizado.
- *Uso de qualquer ferramenta CASE*: Esse critério informa se a abordagem permite que os desenvolvedores possam usar qualquer ferramenta CASE para a criação de modelos. Se a abordagem em si for uma ferramenta CASE, o critério informa se os modelos gerados podem ser usados em outras ferramentas.
- *Transformação automatizada de modelos*: Indica se a transformação de modelos é realizada de forma automatizada, sendo um recurso essencial para não descaracterizar o MDD.
- *Transformação bidirecional de modelos*: Indica se a abordagem é capaz de gerar modelos da direita a partir de modelos da esquerda, e vice-versa.

- *Transformação independente do desenvolvedor*: Indica se a transformação de modelos é realizada sem a intervenção do usuário.
- *Sincronização automatizada de modelos*: Esse critério informa se a sincronização de modelos é realizada automaticamente ou não. Este requisito é essencial para garantir que os modelos estejam sempre consistentes.
- *Sincronização bidirecional de modelos*: Indica se a abordagem é capaz de sincronizar modelos da direita a partir de modelos da esquerda, e vice-versa (*round-trip*).
- *Sincronização independente de desenvolvedor*: Esse critério informa se a sincronização é realizada independentemente da iniciativa do engenheiro de software. Este requisito, juntamente com a sincronização automática, contribui para que os modelos estejam sempre consistentes.
- *Geração de ligações de rastreabilidade durante a transformação*: Informa se as ligações de rastreabilidade (registros de transformação ou rastros) são geradas durante a transformação. As ligações de rastreabilidade são importantes para o processo de sincronização, além de servir de informação para a recuperação de elementos inter-relacionados, possibilita a recuperação de versões anteriores de modelos inter-relacionados e permite a realização de análise de impacto das alterações.
- *Controle de versões de modelos*: Informa se a abordagem gera novas versões dos modelos que foram modificados e se fornece suporte às operações básicas de um sistema de controle de versões (identificação de conflitos, junção, registro de informações sobre a modificação – quem, por que e quando).
- *Suporte ao desenvolvimento distribuído*: Este critério indica se a abordagem fornece algum suporte para que desenvolvedores geograficamente distribuídos possam trabalhar no mesmo projeto. É importante notar que os sistemas de controle de versão servem como suporte ao desenvolvimento distribuído, mas nem todo sistema distribuído fornece suporte ao controle de versões.

3.3 Abordagens Existentes na Literatura

As abordagens selecionadas são apresentadas nas subseções a seguir. A busca foi realizada, principalmente, nos sítios da ACM, IEEE e *Google*. Houve a preocupação com a busca de soluções acadêmicas (*estado da arte*) e comerciais (*estado da prática*). A seleção foi realizada de acordo com os seguintes critérios:

- Estar dentro do contexto do Desenvolvimento Dirigido por Modelos (MDD).
- Ter relação com o controle da evolução de modelos, sincronização ou ao versionamento de modelos. Abordagens que tratam da sincronização de modelos têm como finalidade manter a consistência dos modelos inter-relacionados enquanto evoluem. Abordagens do versionamento de modelos estão dentro do contexto da gerência de configuração, que possui como finalidade controlar a evolução do software ao longo do tempo. Além disso, sistemas de controle de versão possibilitam o desenvolvimento distribuído.
- Usar a abordagem de transformação elaboracionista, quando o trabalho possuir relação com a transformação de modelos.
- Ter sido implementada.

3.3.1 C-SAW

A abordagem C-SAW (LIN, 2005; GRAY *et al.*, 2006) define uma linguagem de alto nível para a especificação de transformações e um *plugin*[10] para o GME[11] (*Generic Modeling Enviromnent* – Ambiente de Modelagem Genérico), uma ferramenta baseada na UML para a criação de ambientes de modelagem para domínios específicos. As tarefas para a construção e evolução de modelos são especificadas como transformações, usando uma linguagem criada pelo próprio autor, chamada de ECL (*Embedded Constraint Language* – Linguagem de Restrições Embutidas).

Para realizar a transformação, o desenvolvedor deve informar os arquivos contendo as regras de transformação a serem aplicadas no modelo aberto no GME. Portanto, o processo de transformação, apesar de ser realizado automaticamente, depende da iniciativa do engenheiro de software. Além disso, a transformação é realizada em apenas uma direção, não são geradas ligações de rastreabilidade e não há

[10] Um *plugin* é uma extensão para outro aplicativo.

[11] Pode ser acessado em http://www.isis.vanderbilt.edu/Projects/gme/

sincronização de modelos inter-relacionados. A abordagem está focada apenas na transformação evolutiva de modelos, não havendo preocupação com a manutenção do histórico das versões de cada modelo, nem com o desenvolvimento distribuído. Além disso, apesar de usar a UML, a transformação não é baseada no QVT.

3.3.2 Abordagem Proposta por Xiong et al

XIONG *et al.* (2007) propuseram uma abordagem automática para fazer a sincronização bidirecion al de modelos baseados no *Ecore*[12] (BUDINSKY *et al.*, 2003) através de transformações definidas a partir da ATL (*Atlas Transformation Language*) (JOUAULT *et al.*, 2005). A sincronização é realizada independente da iniciativa do engenheiro de software. O sistema de sincronização é uma extensão da máquina virtual da ATL. Na abordagem proposta, as entradas para a sincronização são: o modelo-fonte original, o modelo-fonte modificado, o modelo-alvo e a transformação a ser aplicada.

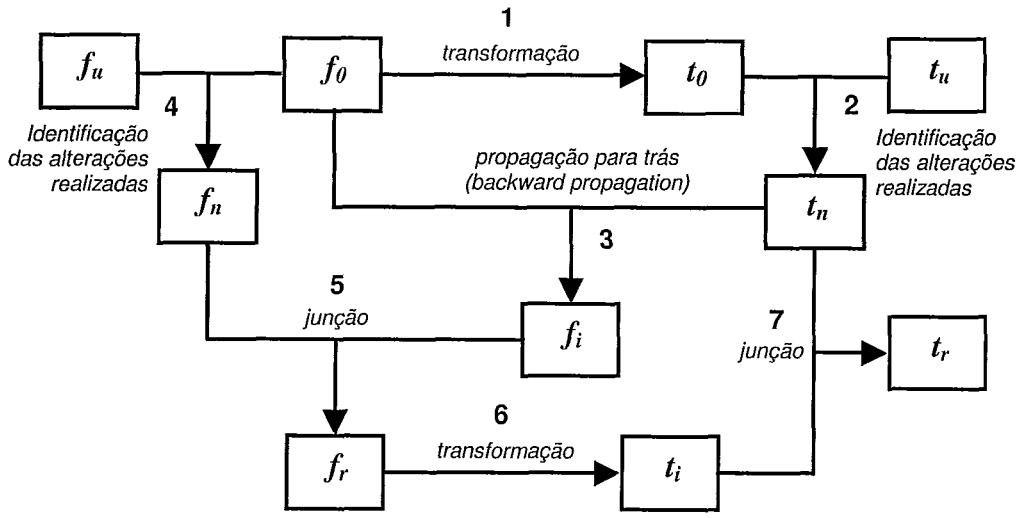

Figura 3.1: Algoritmo de sincronização. Adaptado de XIONG *et al.* (2007)

A sincronização é realizada através do seguinte algoritmo (Figura 3.1): (1) o modelo-alvo original (t_o) é gerado a partir do modelo-fonte original (f_o) através da transformação; (2) o modelo-alvo original (t_o) é comparado com a cópia modificada (t_u), sendo criado um novo modelo-alvo (t_n), contendo marcações que indicam as alterações realizadas (inclusão, alteração e exclusão); (3) é criado um modelo-fonte intermediário (f_i) a partir do modelo-fonte original (f_o) e as alterações do modelo-alvo modificado (t_u)

[12] O Ecore é compatível com o MOF.

são propagadas para este modelo; (4) o modelo-fonte original (f_o) é comparado com a cópia modificada (f_u), sendo criado um novo modelo-fonte (f_n), contendo marcações que indicam as alterações realizadas (inclusão, alteração e exclusão); (5) o modelo-fonte final (f_r) é gerado a partir da junção do novo modelo-fonte (f_n) (possui as marcações das modificações realizadas no modelo-fonte) com modelo-fonte intermediário (fi) (possui as marcações das modificações provenientes do modelo-alvo modificado); (6) a transformação é aplicada no modelo-fonte final (f_r) para gerar o modelo-alvo intermediário (t_i); (7) finalmente, o modelo-alvo final (t_r) é gerado a partir da junção do modelo-alvo modificado (t_n) com o modelo-alvo intermediário (t_i). Neste último passo, alterações que não foram propagadas para o modelo-fonte intermediário (f_i) (passo 3) são transferidas para o modelo-alvo final (t_r). Assim, são gerados um novo modelo-fonte e um novo modelo-alvo sincronizados.

A abordagem é capaz de fazer sincronização automática e bidirecional de modelos UML especificados a partir do meta-metamodelo *Ecore*. Como é possível observar no parágrafo anterior, a sincronização é realizada com o auxílio da transformação de modelos. Contudo, o processo depende da iniciativa do desenvolvedor, além de não gerar ligações de rastreabilidade. Assim como a C-SAW, a abordagem não está voltada para o controle da evolução dos modelos ao longo do tempo, nem fornece suporte para o desenvolvimento distribuído. Além disso, a transformação não é baseada no QVT.

3.3.3 SMoVER

O SMoVER (*Semantically Enhanced Model Version Control System* – Sistema de Controle de Versão Aumentado Semanticamente) (ALTMANNINGER, 2007; ALTMANNINGER *et al.*, 2007; SMOVER, 2008) é um sistema de controle de versão para modelos que considera não apenas a estrutura sintática para a identificação de conflitos, mas também a semântica[13] dos modelos. Os objetivos são descartar conflitos sintáticos que não são relevantes, diminuindo a quantidade de relatos de conflitos, e verificar a existência de conflitos que não tem como serem localizados no nível sintático. Por exemplo, a modificação concorrente do nome de um atributo gera um

[13] No trabalho mencionado, a semântica pode representar o significado dos elementos, atributos e referências uns em relação aos outros, assim como estruturas equivalentes e elementos que interferem em outros elementos mesmo que sejam independentes.

conflito no nível sintático, mas poderia ser descartado no nível semântico, fazendo com que o conflito sintático fosse desconsiderado. Do mesmo modo, a troca concorrente de uma estrutura por outra equivalente poderia não gerar conflito. Seria o caso, por exemplo, da substituição de um nó de decisão por um nó de condição no diagrama de seqüência. Por xoutro lado, algumas alterações podem não gerar conflitos sintáticos, mas, como possuem algum tipo de relação, é possível a ocorrência de conflito semântico. É o caso, por exemplo, quando a mudança da representação de um comportamento pode ter impacto em outras partes do modelo.

A semântica é representada em um metamodelo. Para fazer a identificação de conflitos semânticos, uma transformação é aplicada às versões do modelo que estão sendo usadas para a geração de uma nova versão. Em seguida, é realizada a verificação da existência de conflitos sintáticos e semânticos.

A abordagem proposta utiliza formatos e linguagens padronizadas e faz o versionamento de modelos. Como possui uma arquitetura cliente-servidor, permite o desenvolvimento distribuído. Contudo, como é uma abordagem voltada apenas para o controle de versão de modelos, não é realizada a sincronização de modelos. A transformação é realizada apenas para gerar a visão semântica do modelo, e não para gerar outros modelos. Por esse motivo, este recurso não está sendo levado em consideração para a análise comparativa.

3.3.4 Abordagem Proposta por Alanen

ALANEN (2002) propôs um repositório baseado no MOF com recursos para o versionamento de modelos. O repositório foi criado com base no SMW (*System Modeling Workbench*) (PORRES, 2002), um ambiente para a criação e manipulação de modelos baseados em MOF. Na abordagem de Alanen, os modelos versionados são mantidos em um banco de dados relacional e o acesso ao repositório é realizado através do XML-RPC (XML *Remote Procedure Call*) (XML-RPC, 2008), sendo o transporte do modelo realizado a partir de documento XMI.

A arquitetura cliente servidor da abordagem possibilita o desenvolvimento distribuído. O uso do XMI possibilita a utilização de qualquer ferramenta CASE que exporta e importa modelos usando esse padrão. Contudo, como a abordagem não está voltada para a transformação e sincronização de modelos, esses requisitos não são atendidos.

3.3.5 Abordagem Proposta por Sriplakich et al

A abordagem proposta por SRIPLAKICH *et al* (2008) tem como finalidades solucionar problemas de escalabilidade e manter a consistência de relacionamentos existentes entre diferentes modelos de um projeto MDD, desenvolvido em um ambiente colaborativo, onde os desenvolvedores podem fazer modificações nos modelos concorrentemente. O processo de junção de um modelo modificado com a versão existente no repositório procura identificar as ligações do modelo com outros modelos e fazer as devidas correções, como, por exemplo, excluir ligações de rastreabilidade quando um elemento é excluído.

A abordagem é baseada no ModelBus (SRIPLAKICH, 2007), que utiliza o CVS para armazenar o histórico de versões de modelos. O acesso e atualização dos modelos são realizados através de dois componentes que atuam como intermediários entre o CVS e ferramentas CASE: o *Workspace Manager* e o *Tool Adapter*. O primeiro componente permite a recuperação e atualização de modelos através de operações de *check-out* e *check-in* (*commit*). O componente também é responsável pelo processo de junção de modelos e possibilita a interoperabilidade com as ferramentas através do compartilhamento de arquivos XMI. O segundo componente é uma API que permite a integração de ferramentas CASE com o ambiente ModelBus.

Apesar de utilizar um sistema de controle de versão e operações relacionadas, como a identificação de diferenças e junção de modelos, a abordagem não se propõe fazer a transformação e sincronização de modelos.

3.3.6 Odyssey-VCS

O Odyssey-VCS (OLIVEIRA *et al.*, 2005) é uma ferramenta para o controle de versão de modelos UML, sendo capaz de realizar o versionamento em granuralidade fina. É possível configurar a unidade de comparação e de versionamento. Por exemplo, quando o elemento `classe` é configurado como unidade de versionamento e o `atributo` apenas como unidade de comparação, sempre que um atributo é modificado, a classe recebe uma nova versão. Contudo, como o atributo não é unidade de versionamento, a alteração não gera uma versão desse elemento. A junção e identificação de conflitos são realizadas a partir de um algoritmo Diff3 (MYERS, 1986), que considera o modelo que um usuário está enviando para o repositório (*check-in*), a

versão imediatamente anterior à do modelo do usuário, e a versão mais recente do repositório, que pode conter alterações de outro usuário.

Além do versionamento, o sistema é capaz de fazer a junção de modelos não conflitantes e notificar ao usuário a existência de conflitos. O aplicativo permite a interação com qualquer ferramenta CASE que seja capaz de exportar e importar modelos através do XMI. Contudo, o aplicativo não tem como finalidade fazer a transformação e sincronização de modelos.

3.3.7 Odyssey-MDA

O Odyssey-MDA (MAIA, 2006) é uma abordagem para a transformação de modelos UML, com recurso para a realização de sincronização de modelos. Para fazer a transformação, o engenheiro de software define o modelo da esquerda e da direita, seleciona a especificação de transformação a ser aplicada e a direção da transformação. Se a transformação for da esquerda para a direita, o modelo-fonte é o modelo da esquerda e o modelo-alvo é o modelo da direita, e vice-versa.

A transformação é realizada automaticamente pelo Odyssey-MDA. Porém, depende da iniciativa do desenvolvedor e não gera ligações de rastreabilidade entre os modelos. Apesar de haver sincronização, a identificação de modelos é realizada com base no tipo e nome dos elementos, o que pode gerar alguns problemas, conforme será explicado no Capítulo 4. A abordagem não se propõe a fazer o controle de versão de modelos e não fornece suporte para o desenvolvimento colaborativo. Além disso, a transformação não é baseada no QVT. Os modelos são importados e exportados a partir do XMI, o que possibilita o uso de qualquer ferramenta CASE.

3.3.8 Together

O Together™ (FONG, 2006) é uma ferramenta CASE que fornece recursos para o MDA, tendo suporte para transformações modelo-modelo e modelo-texto. A ferramenta utiliza os padrões da OMG, como o MDA, UML e QVT. Para fazer a transformação, o engenheiro de software deve especificar o modelo-fonte e o modelo-alvo, assim como a especificação de transformação a ser usada. Desse modo, a transformação é automática, mas depende da iniciativa do desenvolvedor. Para algumas plataformas, a transformação de modelos pode ser realizada em duas direções, mas não existe suporte para a sincronização de modelos. Ligações de rastreabilidade são geradas durante a transformação.

Como o Together™ usa o XMI para possibilitar a interoperabilidade de modelos, tornando possível o uso de outras ferramentas CASE. Contudo, a ferramenta não se propõe a fazer o controle de versão de modelos e não fornece suporte ao desenvolvimento distribuído[14].

3.3.9 Enterprice Architect

O *Enterprice Architect* (EA) (TRUYEN, 2005; SPARX, 2008) é um ambiente de modelagem que possibilita a geração de PSMs a partir de um PIM e de código fonte a partir de PSM. Desse modo, está dentro do contexto da abordagem elaboracionista do MDD. A ferramenta permite a definição de transformações, contudo, usando uma linguagem proprietária.

Para fazer a transformação de um modelo, o desenvolvedor deve marcar o PIM com estereótipos e etiquetas, em seguida deve selecionar a transformação a ser aplicada para gerar o PSM. O mesmo procedimento deve ser realizado para a geração do código fonte.

Os PSMs são automaticamente sincronizados com o respectivo PIM sempre que este modelo é modificado. Contudo, não é possível sincronizar o PIM a partir de um PSM. Portanto, a sincronização ocorre apenas em uma direção.

O EA permite o uso de um sistema de controle de versão para controlar a evolução de modelos como, por exemplo, o Subversion (COLLINS-SUSSMAN *et al.*, 2004). Quando o usuário faz o *check-in*, o EA exporta o modelo para um arquivo XMI e envia para o sistema o SCV. O processo inverso é realizado quando o usuário faz o *check-out*. Contudo, o versionamento é baseado nas linhas do arquivo XMI e não na estrutura do modelo. Por esse motivo, o suporte ao versionamento de modelos dessa ferramenta não está sendo levado em consideração.

Além da possibilidade de integrar o EA a um SCV, também é possível configurar um ambiente para o desenvolvimento colaborativo.

3.3.10 Outras Abordagens

Além das abordagens descritas nas subseções anteriores, MATESON *et al* (2004) propuseram uma abordagem centrada em um repositório para permitir a

[14] O versionamento e o desenvolvimento distribuído podem ser realizados com o auxílio do *StarTeam*, outra ferramenta do mesmo fabricante.

evolução controlada de artefatos no contexto do MDD e do AOM (*Aspect Oriented Modeling* – Modelagem Orientada a Aspectos). Essa abordagem possui as seguintes finalidades: (1) fornecer recursos para a persistência e integridade dos artefatos; (2); possibilitar o uso de diversas ferramentas através do XML ou XMI para a troca de dados e suporte a diferentes protocolos para a comunicação; (3) manter os relacionamentos entre os diferentes artefatos; (4) manter um histórico das versões de todos os artefatos e os respectivos relacionamentos; e (5) possibilitar a propagação de modificações para outros artefatos (sincronização). Apesar da abordagem proposta ser bem completa em relação aos critérios de avaliação sugeridos, os seguintes aspectos não são detalhados: direção da transformação (unidirecional ou bidirecional); sincronização dos artefatos e a granuralidade considerada no versionamento de modelos. Além disso, a abordagem não foi implementada até o momento. Por esse motivo, não foi selecionada para a comparação das abordagens.

Outras abordagens abordam a evolução ou a consistência de modelos focando aspectos diferentes. ENGELS *et al.* (2002) propuseram uma abordagem para manter a consistência do UML-RT, uma extensão da UML para modelagem de sistemas de tempo real, mapeando características importantes do modelo para um domínio semântico.

GÎRBA *et al* (2006) desenvolveram a ferramenta *Van* para analisar a evolução de software a partir de modelos recuperados do código fonte de sistemas. Este ferramenta funciona dentro do ambiente de reengenharia *Moose* (GÎRBA et al., 2006). A ferramenta *Van* faz com que o *Moose* passe a manter dados históricos das versões dos modelos. Contudo, as informações sobre o versionamento são adquiridas do CVS (CEDERQVIST, 2005).

CHEN *et al* (2006) propuseram uma abordagem dirigida por modelos formais baseada em transformações de programas e modelos com a finalidade de fazer a reengenharia de sistemas legados, migrando os sistemas para o contexto do desenvolvimento dirigido por modelos.

BARTELT (2008) propôs uma nova abordagem para a junção de modelos, de modo a preservar a consistência com o respectivo meta-modelo.

KÖGEL (2008) propôs um sistema de gerência de configuração para artefatos que possuem estruturas em forma de grafos, mas que não faz referência ao desenvolvimento dirigido por modelos. Outra abordagem para o controle e versão de modelos que não está no contexto do MDD é o Molhado (NGUYEN *et al.*, 2004). Esta

ferramenta é baseada em hipertexto e possui uma extensão para possibilitar o versionamento de modelos UML.

Além das abordagens apresentadas nesta subseção, existe uma outra abordagem digna de nota: AndroMDA (ANDROMDA, 2008). Este software realiza a transformações com base em cartuchos. Um cartucho é equivalente a um mapa de transformação. Apesar de poder realizar transformações modelo-modelo, o principal objetivo é a geração de código fonte através de transformações modelo-texto.

3.4 Análise Comparativa das Abordagens

A Tabela 3.1 classifica as abordagens de acordo com os critérios apresentados na seção anterior. Os campos preenchidos com "✓" indicam os critérios atendidos pela abordagem, enquanto os preenchidos com o símbolo "×" indicam os critérios não atendidos. O símbolo "±" indica que o critério é parcialmente atendido, como, por exemplo, uso de soluções específicas e apenas alguns padrões da OMG.

A partir da análise da Tabela 3.1, é possível chegar às seguintes conclusões:

- Os padrões da OMG estão sendo usados. Contudo, algumas abordagens fazem uso parcial desses padrões (1, 2, 3, 7 e 9), usando outras definições proprietárias.
- É possível notar que a interoperabilidade de modelos é levada em consideração (2 a 8).
- Algumas abordagens (3, 4, 5 e 6) estão voltadas para controlar a evolução de um modelo ao longo do tempo, utilizando um sistema de controle de versão. A atualização é realizada a partir da junção de uma versão do modelo existente no repositório, normalmente a mais recente, com a versão modificada pelo desenvolvedor. Desse modo, desenvolvedores diferentes podem fazer modificações no mesmo artefato.
- Outras abordagens (2,7 e 9) estão voltadas para controlar a evolução de modelos-inter-relacionados a partir da sincronização. Uma vez que esses modelos são independentes, a sincronização permite o controle da evolução de modelos no espaço. Assim, desenvolvedores com papéis específicos podem modificar modelos diferentes.

- As abordagens não consideram ao mesmo tempo o uso de um sistema de controle de versão e a sincronização de modelos para possibilitar a evolução consistente de modelos inter-relacionados.
- Nem todas as ferramentas são independentes de ferramentas CASE (1) ou possibilitam que seus modelos sejam usados em outros aplicativos (9).

Tabela 3.1: Quadro comparativo das abordagens

Requisitos	Abordagens								
	1	2	3	4	5	6	7	8	9
	C-SAW	Xiong et al	SMoVER	Marcus Alanen	Sriplakich et al	Odyssey-VCS	Odyssey-MDA	Together	Enterprice Architect
Utilização de linguagens e formatos padrões da OMG	±	±	±	✓	✓	✓	±	✓	±
Uso de qualquer ferramenta CASE	×	✓	✓	✓	✓	✓	✓	✓	×
Transformação automatizada de modelos	✓	✓	×	×	×	×	✓	✓	✓
Transformação bidirecional de modelos	×	✓	×	×	×	×	✓	±	×
Transformação independente de desenvolvedor	×	×	×	×	×	×	×	×	×
Sincronização automatizada de modelos	×	✓	×	×	×	×	✓	×	✓
Sincronização bidirecional de modelos (*round-trip*)	×	✓	×	×	×	×	✓	×	×
Sincronização independente de desenvolvedor	×	✓	×	×	×	×	✓	×	✓
Geração de ligações de rastreabilidade	×	×	×	×	×	×	×	✓	×
Controle de versões de modelos	×	×	✓	✓	✓	✓	×	×	×
Suporte ao desenvolvimento distribuído	×	×	✓	✓	✓	✓	×	×	✓

3.5 Considerações Finais

Algumas abordagens relacionadas com o controle da evolução de modelos foram avaliadas segundo os seguintes critérios: (1) utilização de linguagens e formatos padrões; (2) possibilidade de usar qualquer ferramenta CASE; (3) transformação

automatizada de modelos; (4) transformação bidirecional de modelos; (5) transformação independente de desenvolvedor; (6) sincronização automatizada de modelos; (7) sincronização bidirecional de modelos; (8) sincronização independente de desenvolvedor; (9) geração de ligações de rastreabilidade; (10) controle de versões de modelos; e (11) suporte ao desenvolvimento distribuído.

Em função da pequena quantidade de abordagens comparadas, as conclusões apresentadas não podem ser consideradas como verdade absoluta. Contudo, é possível perceber como o controle de evolução de modelos vem sendo tratado. É possível notar que existem dois grupos principais: um focado no controle da evolução de modelos inter-relacionados através de sincronização, e outro focado no controle da evolução de modelos ao longo do tempo.

A partir da avaliação realizada, concluímos que nenhuma abordagem atende completamente aos critérios propostos, principalmente no que diz respeito a manter a consistência espacial e temporal dos modelos. Desse modo, propomos uma nova abordagem que contempla os critérios identificados, conforme apresentada no próximo capítulo.

Capítulo 4 – Odyssey-MEC: *A* Abordagem Proposta

4.1 Introdução

No Capítulo 2, foram apresentados os principais conceitos da Arquitetura Dirigida por Modelos (MDA) e a importância de se controlar a evolução dos modelos criados durante o desenvolvimento de software, segundo o Desenvolvimento Dirigido por Modelos (*Model Driven Development* - MDD). No capítulo 3, foram apresentadas as características e deficiências de algumas abordagens.

Neste capítulo, é apresentada uma nova abordagem para o controle de evolução de modelos computacionais no MDD: o Odyssey-MEC – *Odyssey for Model Evolution Control* (Odyssey para Controle de Evolução de Modelos) (CORRÊA *et al.*, 2008a). A abordagem leva em consideração os seguintes requisitos, definidos a partir da revisão da literatura, conforme apresentado no Capítulo 3: (1) utilização de linguagens e formatos padronizados pela OMG, uma vez que aumentam o potencial de reutilização e facilitam a interoperabilidade entre modelos; (2) uso de qualquer ferramenta CASE; (3) transformação automática de modelos; (4) transformação bidirecional de modelos; (5) transformação independente de desenvolvedor; (6) sincronização automática de modelos; (7) sincronização bidirecional de modelos; (8) sincronização independente de desenvolvedor; (9) geração de ligações de rastreabilidade durante a transformação; (10) controle de versões de modelos; e (11) suporte ao desenvolvimento distribuído.

A proposta abrange a geração, sincronização e versionamento de modelos computacionais dentro do contexto de um ambiente de desenvolvimento distribuído. A seção 4.2 apresenta uma visão geral da abordagem, enquanto a seção 4.3 discute os mapas de transformação. A configuração de um projeto MDD é apresentada na seção 4.4. A seção 4.5 trata da marcação de modelos e a seção 4.6 da transformação automática. As ligações de rastreabilidade são comentadas na seção 4.7. A seção 4.8 trata do controle de versões. A seção 4.9 descreve detalhes da sincronização dos modelos. O compartilhamento de modelos e a integridade do repositório são discutidos nas seções 4.10 e 4.11, respectivamente. Finalmente, as considerações finais são apresentadas na seção 4.12.

4.2 Visão Geral da Abordagem

A Figura 4.1 apresenta uma visão geral da abordagem proposta. A abordagem foi concebida com a finalidade de possibilitar o controle da evolução de modelos computacionais inter-relacionados dentro do contexto do MDD, utilizando o MDA como *framework* e permitindo que o desenvolvedor tenha condições de usar qualquer ferramenta CASE que faça a importação e exportação de modelos através do XMI 2.1. Os dois primeiros passos estão relacionados com a preparação da infra-estrutura necessária para o controle da evolução do projeto. Os passos restantes estão relacionados com a execução do projeto propriamente dito.

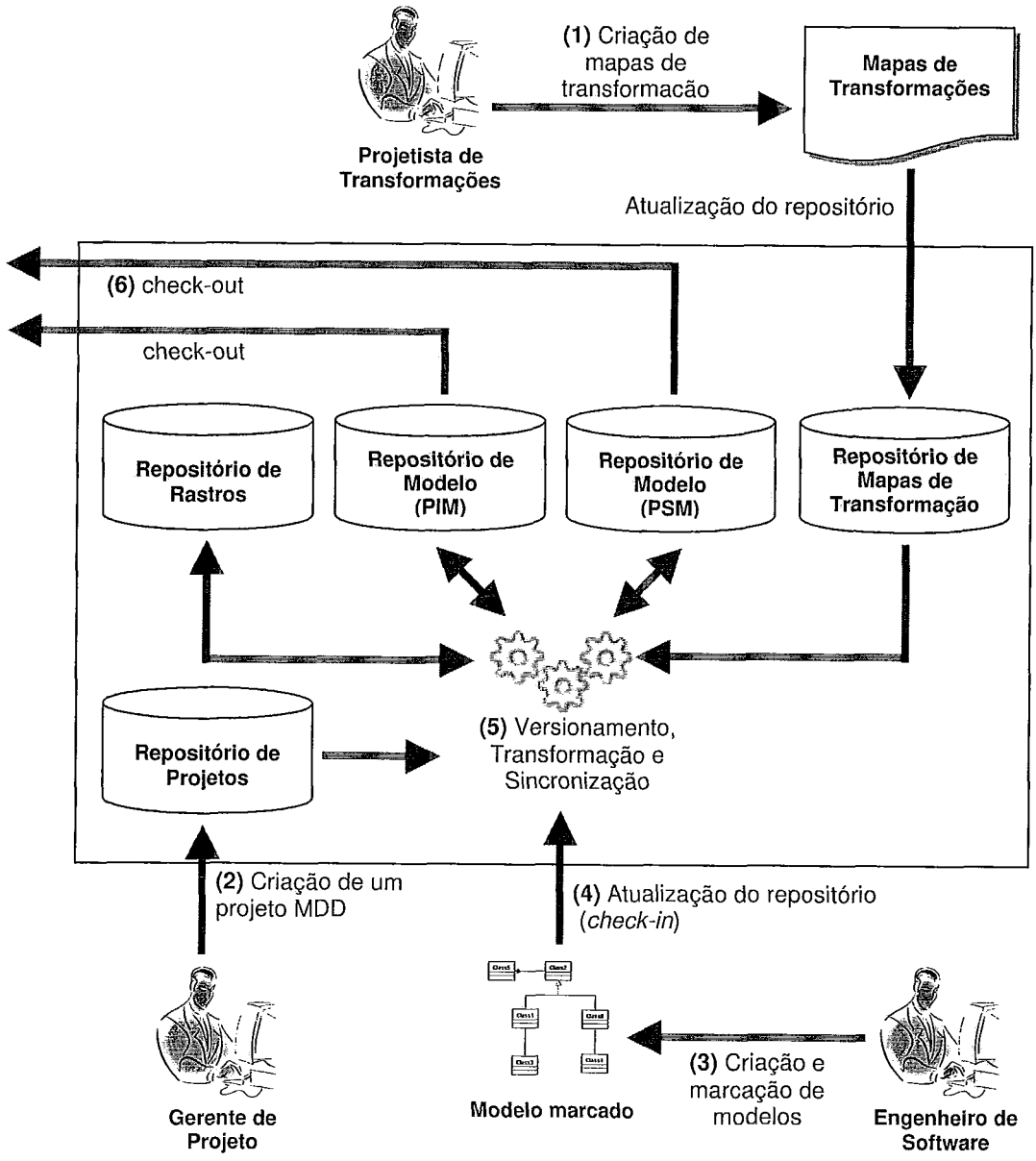

Figura 4.1. Abordagem Odyssey-MEC

Como pode ser observado na Figura 4.1, o **projetista de transformações** cria os mapas de transformação (MT) e os coloca no repositório (1). Cada mapa de transformação define um conjunto de regras de transformação para gerar os elementos do modelo-alvo a partir de elementos do modelo-fonte. Os MTs podem ser definidos para gerar modelos no mesmo nível de abstração ou níveis de abstração diferentes. No contexto do MDA, os mapas de transformação são usados, principalmente, para definir transformações PIM-PSM. Os mapas são usados, posteriormente, para a geração automática de modelos sem a iniciativa do desenvolvedor.

O **gerente de projeto** cria um projeto MDD, informando os modelos e a localização dos respectivos repositórios (2). Em seguida, o gerente define as transformações que devem ser realizadas. Para cada transformação, são informados o **modelo da esquerda**, o **modelo da direita** e o **mapa de transformação**. Do ponto de vista do MDA, o modelo da esquerda, normalmente, é um PIM e o da direita é um PSM. Assim, o Odyssey-MEC pode realizar a transformação e a sincronização automática dos modelos inter-relacionados de um projeto MDD.

O **engenheiro de software** cria um modelo usando uma ferramenta de modelagem e aplica as marcações no modelo (3). Ao fazer o *check-in* do modelo (4), o sistema inicia uma transação, dentro da qual o versionamento, a transformação e a sincronização de modelos são realizados (5). Assim que recebido pelo servidor do Odyssey-MEC, o modelo é versionado. Não havendo conflitos durante o versionamento, é realizado o processo de transformação e sincronização. A máquina de transformações é executada para a criação do modelo da direita, caso o modelo-fonte (modelo de entrada) seja o modelo da esquerda, ou vice-versa. Durante a transformação, as ligações de rastreabilidade são geradas. Após a transformação, o sistema verifica se existe uma versão anterior do modelo-alvo. Se não existir, o modelo passa a ser a primeira versão, caso contrário, a última versão do modelo é lida do repositório e os dados de versionamento são recuperados e colocados no novo modelo-alvo, de modo que o sistema perceba que o modelo é o mesmo do repositório e, assim, gere a nova versão. Esse processo se repete para todos os modelos definidos para o projeto. Depois de versionados e sincronizados, os desenvolvedores podem fazer o *check-out* de qualquer um dos modelos e fazer alterações independentemente (6).

As características da abordagem, definidas com base nos requisitos apresentados no capítulo 2, são apresentadas nas seções a seguir. A transformação e o versionamento

de modelos estão baseados em soluções previamente existentes, uma vez que estão em conformidade com os requisitos a serem atendidos pela abordagem. As implementações dessas soluções são usadas como componentes do Odyssey-MEC.

4.3 Definição de Mapas de Transformação de Modelos

A transformação de modelos é o processo a partir do qual modelo(s)-fonte e mapas de transformação são usados para a geração de modelo(s)-alvo. No contexto do MDA, é possível aplicar uma transformação em um PIM para gerar o PSM de uma plataforma. Contudo, também é possível que o PIM tenha de ser gerado a partir do PSM. Desse modo, é necessário que o mecanismo de transformação seja capaz de realizar a transformação nas duas direções. Como será apresentado 4.9, esse recurso é essencial para o processo de sincronização da abordagem.

O método de transformação escolhido para o Odyssey-MEC é baseado em perfis e uso de padrões e marcações (estereótipos e valores etiquetados) (OMG, 2003b). Os perfis determinam as marcações que podem ser aplicadas aos elementos do modelo. Essas marcações são usadas como referência para a realização da transformação. Os padrões determinam os mapas de transformação que devem ser aplicados. Desse modo, a regra a ser aplicada para a geração de elementos do modelo-alvo é escolhida com base na marcação do elemento do modelo de entrada. Por exemplo, um padrão pode determinar que uma classe com o estereótipo `<<Entity>>` implique na geração de uma classe `EntityBean` para o PSM da plataforma JEE.

O uso de um método baseado em padrões e marcações requer a definição dos mapas de transformação a serem usados pelo dispositivo de transformação. Por exemplo, a Figura 4.2 apresenta um mapeamento de um PIM para um PSM da plataforma JEE.

Para atender a esses requisitos, foi adotada a abordagem proposta para o *Odyssey-MDA* (MAIA, 2006). Nesta abordagem, os mapas de transformação são especificados em um arquivo XML e devem estar de acordo com o esquema XML (*XML schema*) definido para o modelo de transformação (Figura 4.3). Cada mapeamento define o tipo do elemento da esquerda e da direita, os critérios de busca de elementos, a configuração do mecanismo de transformação e os relacionamentos que devem ser gerados no modelo-alvo. Além disso, mapeamentos podem conter outros, de modo a permitir a criação de elementos compostos. Por exemplo, o mapeamento para a construção de uma

classe pode ter mapeamentos para a criação de atributos e métodos. Esses aspectos do mapeamento são descritos nas subseções a seguir.

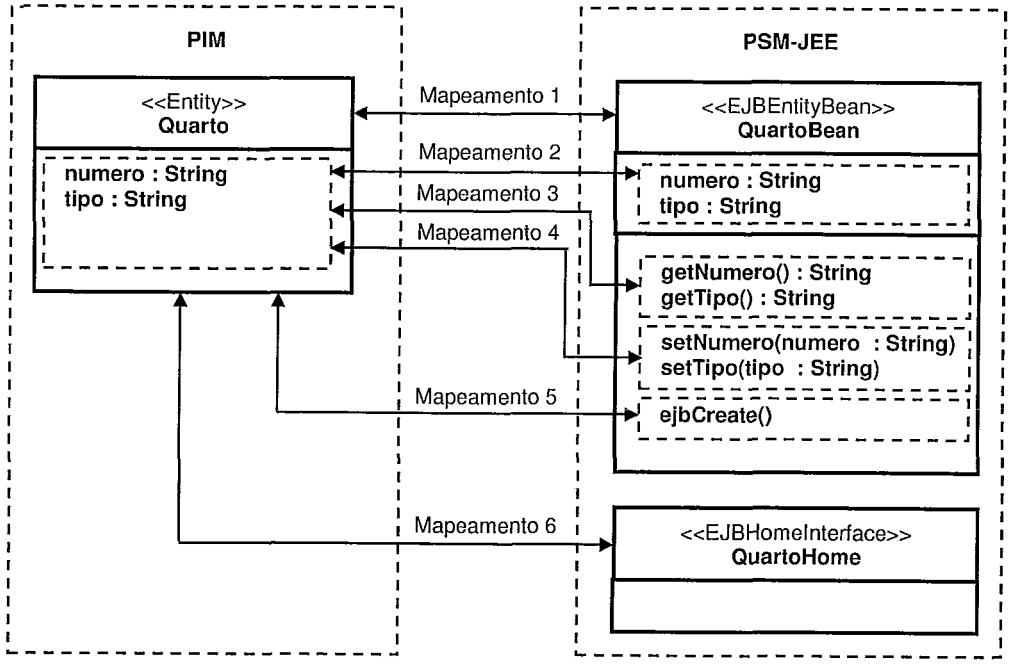

Figura 4.2: Exemplo de mapeamento

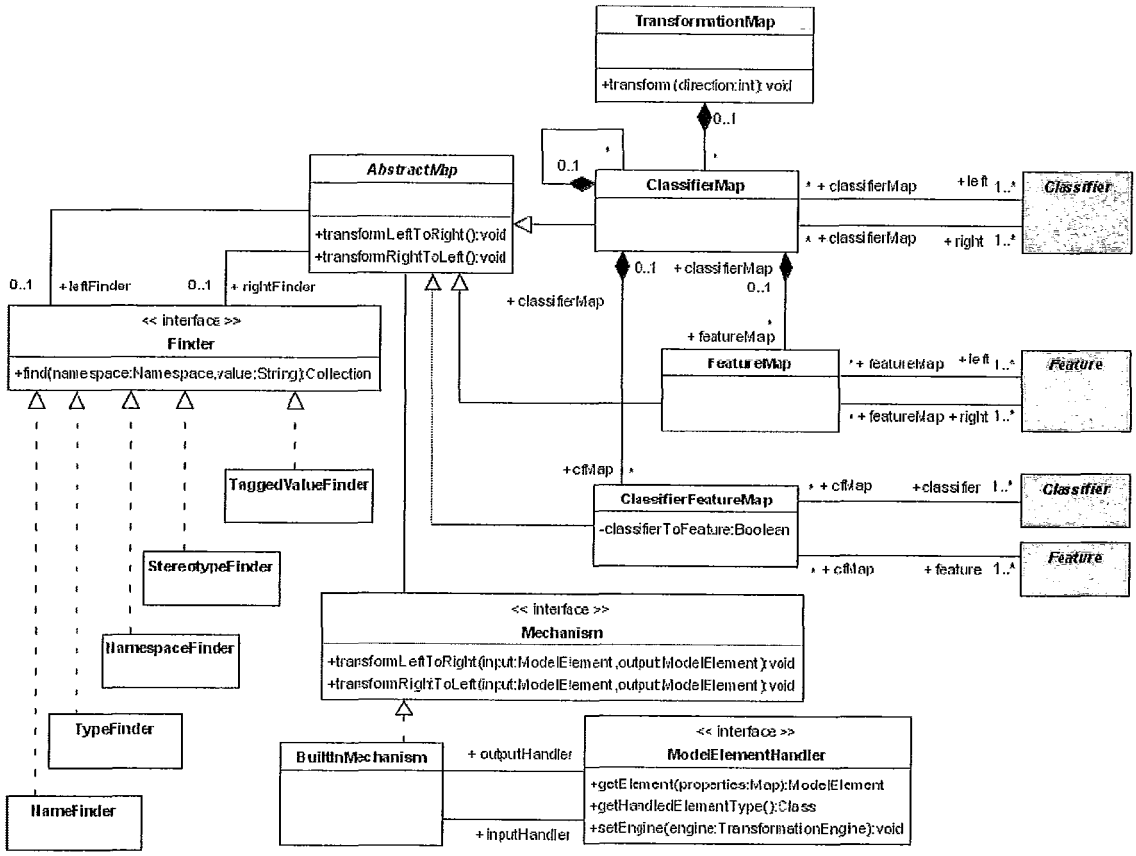

Figura 4.3. Modelo de transformações do Odyssey-MDA (MAIA, 2006)

4.3.1 Definição de Critérios de Busca

Os **critérios de busca** são usados pelo mecanismo de transformação para selecionar os elementos do modelo-fonte e instanciar os mapeamentos a serem aplicados para a geração dos elementos do modelo-alvo. Esses critérios são (MAIA, 2006): tipo, nome, espaço de nomes, estereótipo e valor etiquetado. O critério **tipo** determina a seleção dos elementos a partir de seus tipos, como classes, por exemplo. O **nome** faz com que o mecanismo de transformação selecione elementos que possuem um determinado nome, como elementos com o nome *"create"*. O **espaço de nomes** (*namespaces*) possibilita a seleção de elementos que pertencem ao mesmo espaço de nomes. Um exemplo seria *"br.ufrj.cos.lens.odyssey.odyssey-mec"*. O **estereótipo** determina a seleção de elementos que possuem um estereótipo específico, como classes que possuem o estereótipo <<entidade>>. Finalmente, a **etiqueta** (*tagged value*) possibilita a seleção de elementos que possuem uma determinada etiqueta com um valor específico, como, por exemplo, elementos cuja etiqueta *"persitent"* tenha o valor *"true"*. Esses critérios podem ser usados em conjunto em um mapeamento de modo a tornar a busca mais seletiva. Além disso, os critérios são definidos juntamente com a direção em que devem ser aplicados, como pode ser observado na Figura 4.4.

```
<classifier-map id="map1" name="Entity-Class-PIM - EntiyBean-Class-EJB" ...>
  <finder side="left"  criteria="type"        value="Class" />
  <finder side="left"  criteria="stereotype"  value="entity" />
  <finder side="right" criteria="type"        value="Class" />
  <finder side="right" criteria="stereotype"  value="EJBEntityBean" />
</classifier-map>
```

Figura 4.4: Especificação de critérios de busca (MAIA, 2006)

4.3.2 Definição de Mecanismos de Transformação

O Odyssey-MDA utiliza **mecanismos de transformação** para gerar os elementos do modelo-alvo. Esses mecanismos são instanciados e executados de acordo com o tratador de transformação definido para o elemento da esquerda, o tratador de transformação definido para o elemento da direita e a configuração da transformação especificada em cada mapeamento de transformação. Por exemplo, um mapeamento que informa um *tratador de classe* para o elemento da esquerda e um *tratador de interface* para o elemento da direita faz com que a máquina de transformação crie uma instância do mecanismo de transformação do tipo Classe-Interface. Os tratadores disponibilizados

pelo Odyssey-MDA possuem os mesmos nomes que os elementos que tratam: *type*[15], *class*, *interface*, *attribute*, *operation* e *component*. Além desses, é possível estender o Odyssey-MDA para fazer a transformação de outros tipos de elementos através de *plug-ins*[16] (BIRSAN, 2005). Na transformação esquerda-direita, o mecanismo de transformação utiliza o tratador do elemento da direita, caso contrário, utiliza o tratador do elemento da esquerda.

Além da definição dos mecanismos de transformação, é possível definir um conjunto de propriedades que podem ser usadas para a configuração do elemento gerado. Por exemplo, é possível informar que o elemento criado deve receber o estereótipo `<<EJBHomeInterface>>`. Além disso, é possível informar a direção de transformação em que a propriedade deve ser aplicada. A Figura 4.5 apresenta um exemplo de configuração para a execução de um mapeamento.

```
<classifier-map id="map5" name="Entity-Class-PIM-EntityBean-HomeInterface-EJB">
  <property name="left_type"  value="Class" />
  <property name="right_type" value="Interface" />
  <property name="stereotype" value="EJBHomeInterface" direction="forward" />
</classifier-map>
```

Figura 4.5: Propriedades de configuração do mapeamento de transformação (MAIA, 2006)

4.3.3 Definição de Criação de Relacionamentos

A última configuração de um mapeamento a ser realizada é determinar os relacionamentos que devem ser gerados. Esses relacionamentos podem ser criados a partir de relacionamentos existentes no modelo-fonte ou em função da transformação. No primeiro caso, é necessário declarar o atributo `preserve-relationships` no mapa de transformação, conforme apresentado na Figura 4.6.

```
<classifier-map id="map1" name="Map Name" preserve_relationships="true">
...
</classifier-map>
```

Figura 4.6: Propagação de relacionamento

[15] O tratador de tipos simples foi criado para permitir a transformação e sincronização de tipos simples, como será explicado mais adiante.

[16] *Plug-in* é um mecanismo que permite estender as funcionalidades de um programa sem que seja necessário compilá-lo novamente.

No segundo caso, é necessário especificar o relacionamento a ser gerado, os mapas de transformação que foram usados para a geração dos elementos que fazem parte do relacionamento, e as propriedades necessárias para especificar como os elementos se relacionam. A Figura 4.7 apresenta um exemplo para a criação de uma generalização. Neste exemplo, a propriedade `parent` informa que o elemento gerado pelo mapeamento `map1` faz o papel de super classe, enquanto a propriedade `child` informa que o elemento gerado pelo mapeamento `map2` é a subclasse do relacionamento.

```
<classifier-map id="map1" name="Map 1">
...
</classifier-map>
<classifier-map id="map2" name="Map 2">
...
</classifier-map>
<relationship type="Generalization" direction="forward">
  <map id="map1">
    <property name="parent" value="yes" />
  </map>
  <map id="map2">
    <property name="child" value="yes" />
  </map>
</relationship>
```

Figura 4.7: Criação de um relacionamento de generalização

Os relacionamentos que podem ser criados a partir do Odyssey-MDA são: generalização (*generalization*), associação (*association*), dependência (*dependency*), ligação (*binding*), abstração (*abstraction*), uso (*usage*) e permissão (*permition*). As propriedades a serem usadas variam de acordo com o tipo de relacionamento. As propriedades são:

- *child* e *parent*, para gereralização;
- *client* e *supplier*, para dependência, ligação, abstração, uso e permissão;
- *is_navigable*, *ordering*, *aggregation*, *multiplicity* e *changeability*, para associação.

4.3.4 Mapa de Transformação para Tipos Simples

O PIM e o PSM podem ter tipos simples diferentes. Por exemplo, o tipo inteiro pode ser definido como *Integer* no PIM e como *int* no PSM para a plataforma JEE. Para

que tipos simples sejam mapeados corretamente, foi criado um mapa de transformação que permite ao mecanismo de transformação criar os tipos simples correspondentes no modelo-alvo.

É possível que os elementos que representam tipos simples sejam mantidos separadamente do modelo. Desse modo, o modelo passa a ter referências para elementos externos. Isso faz com que as ferramentas que usam o mesmo modelo precisem conhecer a localização desses elementos. Assim, tanto as ferramentas quanto os modelos tornam-se dependentes dessa localização. Além disso, conforme descrito no parágrafo anterior, os modelos podem ter tipos diferentes. Levando estes dois aspectos em consideração, foi definido que os elementos simples usados pelos modelos, cuja evolução será controlada pelo Odyssey-MEC, devem estar dentro de um pacote de tipos simples localizado no próprio modelo.

4.3.5 Mapas de Transformação no Odyssey-MEC

Esta seção apresentou os mapas de transformação do Odyssey-MDA como recurso para a especificação das transformações. Os mapas são criados pelos projetistas de transformações e são mantidos em arquivos XML.

No contexto do Odyssey-MEC, o projetista mantém os mapeamentos no repositório de mapas de transformações, a partir do qual são acessados para a realização de transformações e sincronização de modelos. A próxima seção descreve como um projeto MDD é configurado, de modo que a transformação e sincronização sejam realizadas automaticamente, sem a iniciativa do desenvolvedor.

4.4 Configuração de Projetos MDD

Um projeto MDD é caracterizado pela existência de modelos inter-relacionados. A abordagem tem como objetivo propiciar a evolução consistente e automática desses modelos, realizando transformações e sincronizações no lado servidor. Para isso, durante a configuração do projeto, é necessário informar os modelos do projeto e as transformações que devem ser realizadas. Para cada transformação, são informados os modelos da esquerda e da direita, assim como o mapa de transformação a ser empregado.

Um modelo da esquerda pode ser usado para gerar mais de um modelo da direita. Além disso, um modelo-alvo pode ser um modelo-fonte para a geração de outro

modelo-alvo. No exemplo da Figura 4.8, o modelo *A* (PIM) é usado para gerar o modelo *B* (PIM refinado), que é usado para a geração dos modelos *C1* (PSM- JEE) e *C2* (PSM – .NET), no caso de uma transformação esquerda-direita (engenharia adiante). Em uma transformação direita-esquerda (engenharia reversa), o modelo *C2* é usado para a geração do modelo *B*, que é usado para a geração do modelo A, usando uma transformação direita-esquerda, e para a geração de C1, usando uma transformação esquerda-direita.

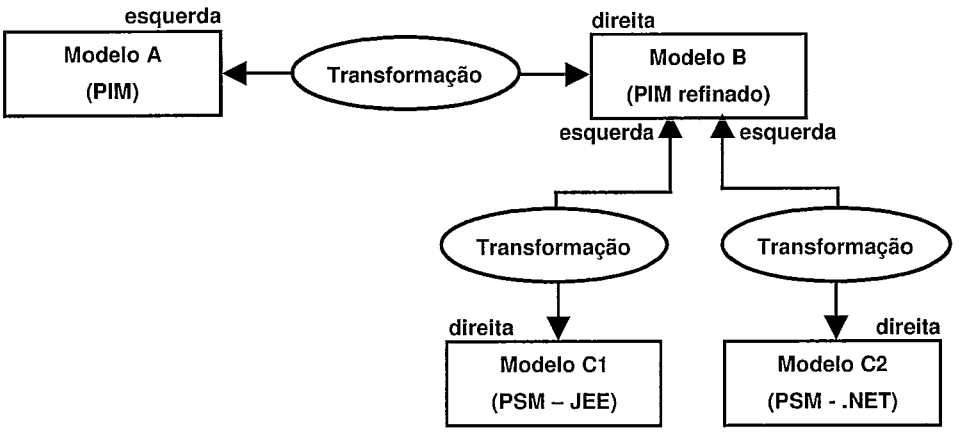

Figura 4.8: Geração de um modelo a partir de outro

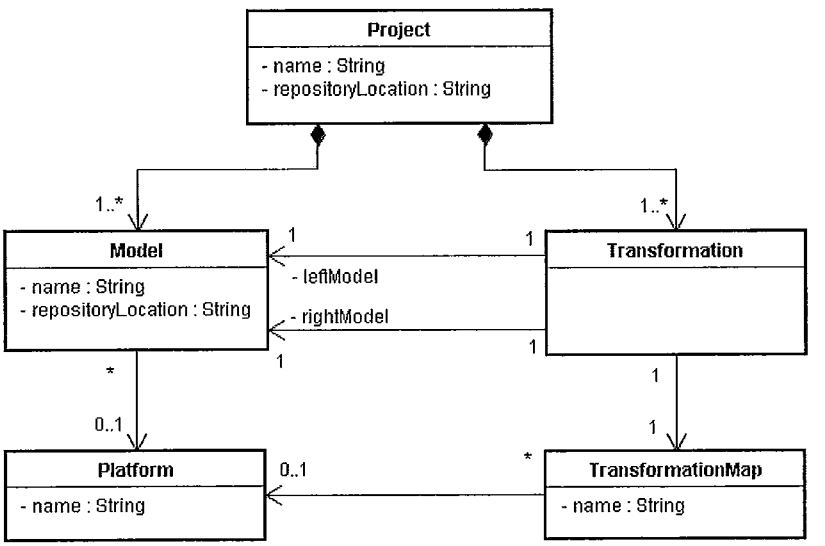

Figura 4.9: Meta-modelo para a especificação de um projeto MDD

Além de relacionar os modelos, é necessário informar o mapa de transformação a ser usado. Para representar um projeto MDD, os modelos e as respectivas transformações, foi criado o meta-modelo apresentado na Figura 4.9. O projeto (*Project*) possui um nome (*name*) e seu repositório tem uma localização

(*repositoryLocation*), onde são armazenadas as ligações de rastreabilidade. O projeto é composto de modelos. Cada modelo (*Model*) possui um nome (*name*), está localizado em um repositório (*repositoryLocation*) e pode estar relacionado a uma plataforma tecnológica (*Platform*). O projeto também possui transformações. Cada transformação faz referência ao modelo da esquerda, ao modelo da direita e ao mapa de transformação a ser usado durante a transformação. O mapa pode pertencer ou não a uma plataforma específica.

4.5 Marcação de Modelos

Os elementos de um modelo devem conter **marcações** (estereótipos e etiquetas), para que a máquina de transformação tenha como selecioná-los em função dos mapeamentos e respectivos critérios e assim gerar os elementos do modelo-alvo. A marcação consiste basicamente na aplicação de estereótipos e etiquetas específicos nos elementos de um modelo. Por exemplo, para a criação de classes *EJBeans* para a plataforma JEE, é necessário marcar as respectivas classes PIM com o estereótipo `<<Entity>>`.

A marcação de modelos UML 2.1 requer o uso de perfis para a representação de estereótipos e etiquetas. O perfil deve ser aplicado ao modelo antes dos estereótipos e etiquetas. Para a marcação de um PIM para um sistema de informação, foi criado um perfil para a identificação de classes de entidades e serviços, além dos perfis para plataformas de software. Esses perfis devem ser usados pelas ferramentas CASE e preservados durante os processos de importação e exportação dos modelos. Para outros tipos de marcações, é necessário criar outros perfis ou estender os existentes.

A marcação dos modelos pode ser realizada com o auxílio da própria ferramenta CASE. Contudo, normalmente, é necessário aplicar as marcações a cada elemento individualmente. Para reduzir o tempo de aplicação de marcações, uma opção mais eficiente é selecionar vários elementos de uma única vez e aplicar as marcações (MAIA, 2006).

4.6 Transformação Automática de Modelos

A transformação de modelos é caracterizada pela criação de um modelo-alvo a partir de um conjunto de modelos-fonte. Para o Odyssey-MEC, a solução adotada foi a realização de transformações baseadas em mapas de transformação e marcações

aplicadas nos elementos dos modelos-fonte, usando como referência a abordagem proposta por MAIA (2006).

Para fazer a transformação de modelos, o Odyssey-MEC informa à máquina de transformação o modelo-alvo, o modelo-fonte, o mapa de transformação e a direção da transformação. A partir dessas informações, a máquina de transformação executa as seguintes operações: (1) instancia os mecanismos necessários para fazer as transformações; (2) seleciona os elementos do modelo-fonte com base nos mapas de transformação e critérios de seleção; e (3) finalmente, gera os elementos do modelo-alvo e as ligações de rastreabilidade[17], que serão detalhadas na seção 4.7.

4.6.1 Não Transformação de Elementos

Durante o desenvolvimento, é possível que um modelo tenha de ser completado manualmente pelo engenheiro de software. No contexto do MDA, por exemplo, pode ser necessário incluir novos métodos no PSM de uma plataforma para torná-lo mais completo para a geração de código fonte. Como são específicos do PSM, não devem ser propagados para o PIM no caso de uma transformação inversa (direita-esquerda). Para isso, o elemento deve ser marcado com a etiqueta *isTransformable* com o valor *false*.

4.7 Ligações de Rastreabilidade

As ligações de rastreabilidade relacionam os elementos do modelo-alvo com os elementos do modelo-fonte que foram usados durante a transformação. No Odyssey-MEC, cada ligação de rastreabilidade relaciona um elemento do modelo da direita com um elemento do modelo da esquerda, além de manter a identificação da transformação que foi realizada (Figura 4.10). Essas ligações são geradas durante a transformação e são usadas pela máquina de sincronização para a localização de elementos do modelo-alvo, como será discutido na seção 4.9 .

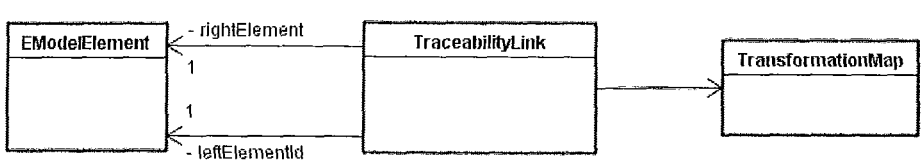

Figura 4.10: Ligação de rastreabilidade

[17] A abordagem de transformação de modelos foi expandida com o objetivo de gerar as ligações de rastreabilidade.

4.8 Controle de Versões de Modelos

O controle de versão de modelos é caracterizado pela criação e controle do histórico das versões de cada modelo. Conforme discutido no Capítulo 2, os sistemas de controle de versão baseados em arquivos de texto utilizam a linha como unidade de comparação e o arquivo como unidade de versionamento. Esses sistemas não levam em consideração a estrutura e a semântica dos elementos contidos no arquivo. Desse modo, toda vez que uma linha é modificada, o arquivo recebe uma nova versão. Portanto, não são adequados para manter o histórico de versões de modelos.

Um aspecto relevante para a geração de versões de modelos é a **granularidade** considerada. Essa propriedade está relacionada com a composição de elementos a partir de outros (OLIVEIRA, 2005). A Figura 4.11 apresenta níveis de granuralidade do arquivo até o parâmetro do método de uma classe. Neste caso, o arquivo representa a granuralidade mais grossa, enquanto o parâmetro representa a granuralidade mais fina. Quanto mais fina a granuralidade, maior é a quantidade de elementos versionados. Além disso, a criação de uma nova versão de um elemento de granuralidade fina implica sempre na criação de uma nova versão do elemento de granuralidade mais grossa que é composto pelo elemento. Por exemplo, a mudança do nome do atributo de uma classe implica em uma nova versão do atributo e, conseqüentemente, da classe. O uso de uma granuralidade mais fina propicia alguns benefícios, como permitir ao gerente de projeto acompanhar a evolução de elementos específicos durante o desenvolvimento (ex.: saber quando e quem fez alterações em uma classe) e maior facilidade para a junção de elementos. Por outro lado, a maior quantidade de elementos versionados pode prejudicar o desempenho do sistema (OLIVEIRA, 2005).

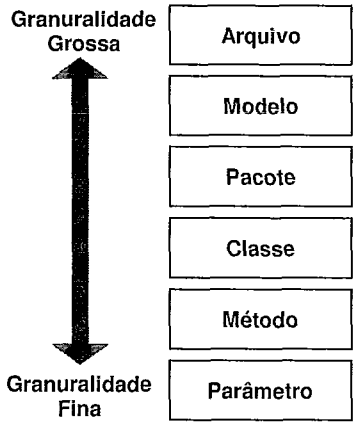

Figura 4.11: Exemplo de variação de granuralidade - Adaptado de OLIVEIRA (2005)

Um dos requisitos da abordagem é manter a evolução de modelos ao longo do tempo. Para isso, é necessário manter um histórico de versões de todos os modelos de um projeto MDD. Para atender a este requisito, foi adotada a abordagem definida para o Odyssey-VCS, que permite o controle de versão de qualquer elemento UML em granularidade fina (OLIVEIRA, 2005; MURTA *et al.*, 2008).

Na abordagem adotada para o Odyssey-VCS, os dados de versionamento são mantidos no repositório separadamente do modelo. Esses dados são estruturados de acordo com o meta-modelo apresentado parcialmente na Figura 4.12. Os principais elementos do meta-modelo de versionamento são: *Project*, *ConfigurationItem*, *Version*, *Transaction, User* e *EModelElement*. O elemento *Project* representa um projeto de *software* e mantém o número da versão atual do projeto (*currentVersionNumber*). O projeto faz referência ao item de configuração raiz (*rootConfigurationItem*), que normalmente representa um elemento *Model* da UML. O elemento *ConfigurationItem* representa um item de configuração, cujo tipo é informado pelo atributo *type*. Exemplos de itens de configuração são classes, componentes, atributos e métodos. Cada item de configuração pode ter várias versões. Desse modo, o elemento *Version* mantém os dados do versionamento de um item de configuração e faz referência ao elemento UML que está sendo versionado (*EModelElement*[18]). Além disso, uma versão pode ter referência para a versão anterior e posterior, assim como para versões que foram ramificadas (*branched*) ou juntadas (*merged*). Os significados dos atributos e associações do elemento *Version* estão descritos em detalhes na Tabela 4.1.

O elemento *Transaction* registra os dados referentes às operações realizadas. O tipo da transação, como *login*, *check-in* e *check-out* é mantido pelo atributo *name*. O atributo *timeStamp* indica a data e a hora da transação, enquanto *description* mantém a descrição da transação informada pelo engenheiro de software. Finalmente, *User* representa um usuário, enquanto a sua associação com *Transaction* informa o usuário que realizou a transação.

---

[18] O elemento *EModelElement* pertence ao meta-metamodelo *Ecore,* usado como referência no framework de modelagem do Eclipse (EMF – *Eclipse Modeling Framework*). No projeto desta dissertação, é utilizado um meta-modelo UML criado com base no Ecore. Desse modo, qualquer elemento UML que seja um *EModelElement* pode ser versionado.

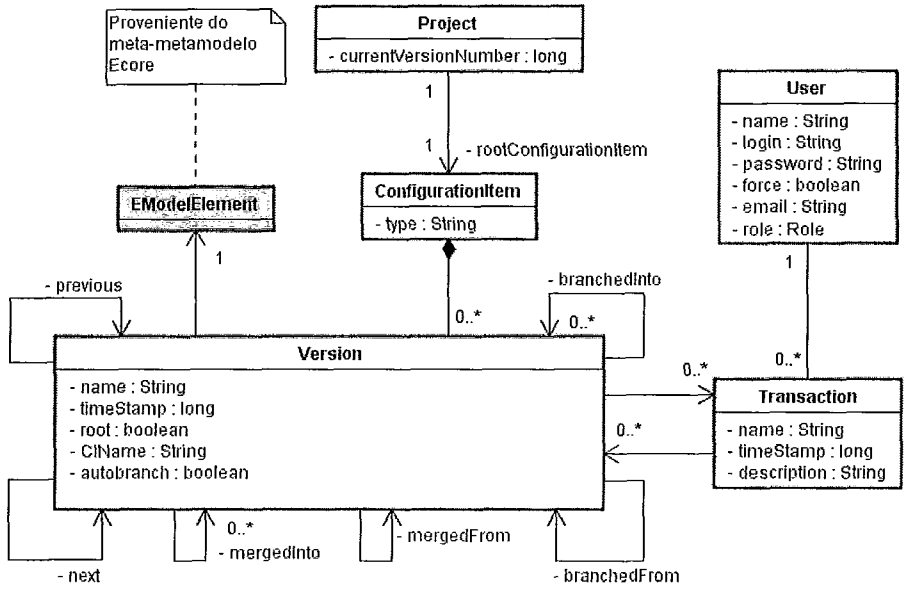

Figura 4.12: Meta-modelo do Odyssey-VCS para o controle de versão

Tabela 4.1: Propriedades a associações do elemento *Version*

Tipo	Nome	Descrição
Atributos	*name*	Número da versão.
	timeStamp	Data e hora da criação da versão.
	Root	Informa se a versão é a raiz (primeira versão).
	ClName	Valor da propriedade *name* do elemento UML.
	autoBranch	Informa se a versão é proveniente do ramo gerado automaticamente pelo Odyssey-VCS para manter o versionamento do modelo de um usuário.
Associações	*next*	Referência para a versão posterior à atual.
	previous	Referência para a versão anterior.
	branchedFrom	Versão anterior a partir da qual a versão em questão foi ramificada.
	branchedInto	Versões anteriores ramificadas a partir da versão em questão.
	mergedFrom	Versão originária da versão atual em função de uma operação de junção
	mergedInto	Versões geradas a partir de junção realizada com a versão atual.
	eModelElement	Referência para o elemento UML que está sendo versionado.
	configurationItem	Referência para o elemento *CondigurationItem.*
	transaction	Referência para as transações que registram as modificações relacionadas à versão.

4.8.1 Identificação e Localização de Elementos

Um aspecto chave para o versionamento é a localização dos elementos do modelo. Por exemplo, quando o usuário realiza o *check-in* de um modelo, o mecanismo de versionamento precisa localizar no repositório os elementos correspondentes aos elementos do modelo do usuário.

Para que os elementos de um modelo possam ser localizados, é necessário alguma forma de identificação. Um modo de se identificar um elemento é a partir da combinação do tipo e nome do elemento, mais o pacote onde o elemento se encontra. Contudo, a alteração do nome ou a remoção do elemento para outro pacote impossibilita a localização do elemento. Uma outra forma de se identificar um elemento é usar um valor único, conforme o conceito de chave primária utilizado em banco de dados. Quando um modelo é mantido em um arquivo XMI, os elementos possuem um atributo chamado *xmiId*, que identifica unicamente cada elemento. Contudo, as ferramentas de modelagem, normalmente, geram novamente esses identificadores quando o modelo é exportado, o que impossibilita o uso do *xmiId* para a localização de elementos no repositório.

Para resolver o problema, a solução adotada para o Odyssey-VCS é semelhante à de alguns sistemas de controle de versão, como o CVS (CEDERQVIST, 2005) e o Subversion (COLLINS-SUSSMAN *et al.*, 2004). Ambos mantém informações de versionamento de arquivos em diretórios localizados no espaço de trabalho do desenvolvedor. Contudo, como os elementos de um modelo manipulado pelo Odyssey-VCS estão contidos em um arquivo, as informações são mantidas no próprio modelo através dos atributos do estereótipo *OdysseyVCS*, aplicado à todos os elementos do modelo. Entre as informações de versionamento, está o valor do *xmiId* inicial do elemento. Desse modo, quando o usuário faz o *check-out*, o identificador permanece no modelo. Quando o usuário faz o *check-in* do modelo após as alterações, o *xmiId* mantido como atributo de *OdysseyVCSClient* (valor etiquetado) é usado para a localização do elemento no repositório. Portanto, esse procedimento possibilita a identificação do elemento original, mesmo que a ferramenta de modelagem tenha criado um novo valor para o *xmiId* do elemento durante a exportação[19].

[19] A ferramenta de modelagem deve preservar as informações inseridas no modelo pelo Odyssey-VCS.

4.8.2 Junção de Modelos

Quando pessoas diferentes realizam o mesmo papel no desenvolvimento de software, existe a possibilidade desses profissionais realizarem cópias do mesmo artefato de software para fazer modificações. Cópias diferentes do mesmo artefato podem trazer transtornos para o desenvolvimento, como, por exemplo, dificuldade para manter e selecionar a cópia correta.

Para evitar a existência de cópias diferentes de um artefato de software, é necessário criar uma nova versão que contenha as modificações realizadas separadamente. Em sistemas de controle de versão, esse procedimento é chamado de junção (*merge*).

Para realizar a junção de modelos, o Odyssey-MEC utiliza a abordagem proposta por MURTA (2008), inspirada no algoritmo diff3 (MYERS, 1986). Nesta abordagem, três versões diferentes de um item de configuração (elemento) são usadas para a criação da nova versão:

- **Versão Base**: É a versão obtida pelo desenvolvedor a partir da operação de *check-out*.
- **Versão Fonte (versão do usuário)**: Versão base modificada pelo desenvolvedor e disponibilizada para o mecanismo de junção através do *check-in*.
- **Versão Alvo (versão atual)**: é a versão mais recente do modelo existente no repositório. Após o usuário ter obtido o modelo a partir do *check-out*, outros usuários podem ter realizado o *check-in* de suas cópias, gerando novas versões posteriores à versão base (Figura 4.13).

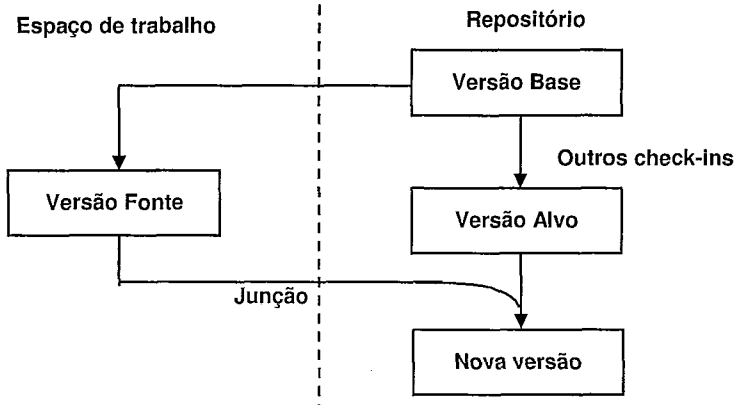

Figura 4.13: Junção de versões diferentes de um item de configuração

O objetivo da abordagem é gerar uma nova versão, fazendo a junção do modelo fonte com o modelo alvo (Figura 4.13), mas usando a versão base como referência para saber quais alterações são conflitantes e quais devem ser mantidas na nova versão.

Tabela 4.2: Resultados da análise de existência – Adaptado de MURTA (2008)

∃ B	∃ F	∃ A	Resultado
V	V	V	O algoritmo de junção continua (ver tabela de processamento de processamento de atributos - Tabela 4.3).
V	V	F	**Se** F == B **então** N = null* **senão** notificar o seguinte conflito: "o fonte foi modificado e o alvo foi excluído"
V	F	V	**Se** A == B **então** N = null* **senão** notificar o seguinte conflito: "alvo modificado e fonte excluído"
V	F	F	N = null*
F	V	V	Impossível (elementos diferentes não podem ter o mesmo identificador)
F	V	F	N = F
F	F	V	N = A
F	F	F	N = null*

**null significa que o elemento foi excluído.*

Legenda: B = Versão Base (versão comum à do usuário e a mais recente)
F = Versão fonte (usuário)
A = Versão alvo(última versão do repositório)
N = Nova versão (criada a partir da junção)

Tabela 4.3: Processamento de atributos – Adaptado de MURTA (2008)

F == B	A == B	F == A	Resultado
V	V	V	N = A (ou N = S ou N = B)
V	V	F	Impossível (transitividade)
V	F	V	Impossível (transitividade)
V	F	F	N = A
F	V	V	Impossível (transitividade)
F	V	F	N = F
F	F	V	N = A (ou N = F)
F	F	F	Notificar conflito: "o mesmo atributo foi alterado nas versões fonte e alvo"

**Ver legenda da tabela 4.2.*

A junção é realizada nas seguintes etapas: (1) análise de existência, (2) processamento de atributos, e (3) processamento de relacionamentos. A **análise de existência** verifica se o algoritmo de junção deve continuar, parar ou notificar conflito. Os possíveis resultados da análise estão relacionados na Tabela 4.2. O **processamento de atributos** realiza a junção dos atributos (propriedades) de cada elemento, conforme a Tabela 4.3. O processamento de relacionamentos verifica as modificações em elementos que estão relacionados com outros. Quando o relacionamento é do tipo contêiner, as alterações no lado oposto do relacionamento determinam uma nova versão do elemento. Por exemplo, alterações em um método implicam na criação de uma nova versão de sua

respectiva classe. Quando o relacionamento não é do tipo contêiner, apenas a remoção ou inclusão de um elemento oposto implica na geração de uma nova versão do elemento. Por exemplo, caso existam duas classes relacionadas, como *Cliente* e *Venda*, alterações em uma classe não implicam em uma nova versão da outra. Contudo, a criação de uma nova classe *Venda* associada a *Cliente* faz com que uma nova versão da classe Cliente seja criada.

4.8.3 Recuperação de Perfis Aplicados aos Modelos

Modelos UML podem ser estendidos a partir do uso de perfis (OMG, 2007). Um perfil representa um conjunto de estereótipos que podem ser aplicados ao modelo, de modo a estender os significados dos elementos. Por exemplo, um estereótipo `<<Entity>>` pode ser usado para informar as classes que representam as entidades do domínio de um problema. Além disso, um estereótipo pode conter atributos que possibilitam estender os dados relacionados aos elementos, como, por exemplo, a identificação da versão do elemento. O processo de junção cria um novo modelo como resultado da junção da última versão existente no repositório com o modelo do usuário. Para que dados relevantes sobre o modelo não sejam perdidos, os perfis existentes no modelo do usuário são transferidos para o modelo gerado pelo mecanismo de versionamento.

4.9 Sincronização de Modelos

O versionamento possibilita o controle da evolução de modelos ao longo do tempo. Contudo, assim como foi discutido no Capítulo 2, esse recurso não é suficiente para garantir a evolução consistente de modelos inter-relacionados, uma vez que os modelos podem evoluir independentemente.

Para que modelos computacionais inter-relacionados possam evoluir de maneira consistente, é necessário mantê-los sincronizados através da aplicação da engenharia *round-trip*. Essa abordagem preconiza a sincronização automática de modelos, preservando as particularidades de cada modelo e propagando as alterações realizadas em elementos que possuem relação com elementos de outros modelos. A relação entre elementos de modelos diferentes é decorrente do processo de transformação. Portanto, a sincronização deve manter os modelos inter-relacionados consistentes entre si e com os respectivos mapas de transformação que foram usados. Por exemplo, se um mapa de

transformação define a criação de uma classe do tipo *EntityBean* a partir de uma classe do tipo *Entity*, e vice versa, a sincronização deve garantir que as classes existam nos respectivos modelos.

O procedimento adotado para a sincronização de modelos é gerar um novo modelo-alvo e fazer a junção desse modelo com a sua versão anterior. Assim, o novo modelo-alvo mantém-se consistente com o modelo-fonte e com os mapas de transformação.

A geração do modelo-alvo é realizada a partir da execução do mecanismo de transformação. Se o modelo-fonte for o modelo da esquerda do mapa de transformação, é executada uma transformação esquerda-direita, caso contrário, é realizada uma transformação direita-esquerda.

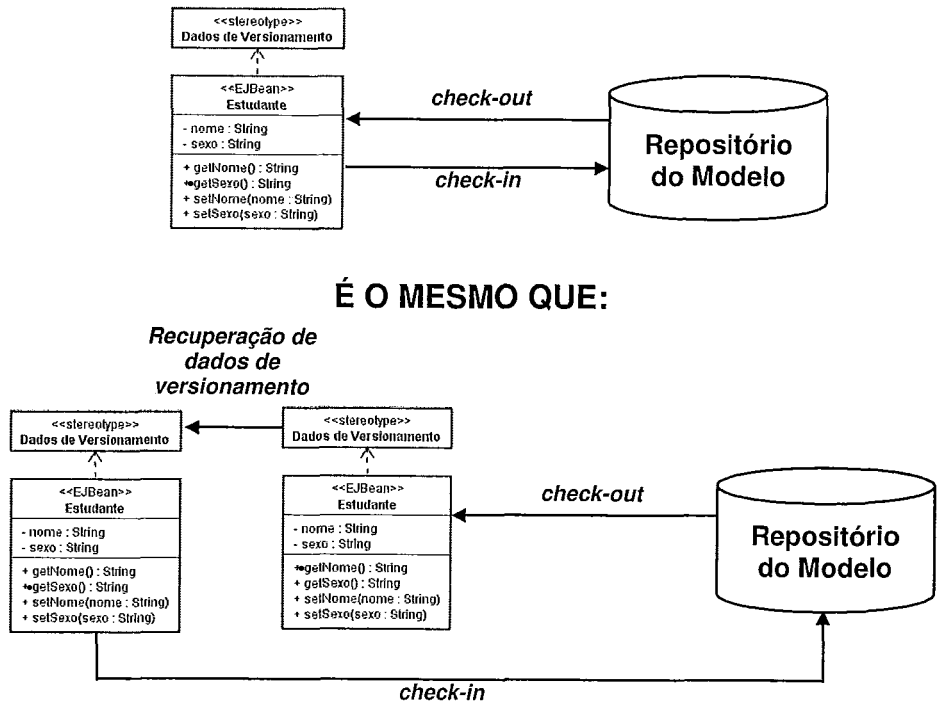

Figura 4.14: Recuperação de dados e versionamento

Depois que o novo modelo-alvo é gerado, é necessário fazer a junção desse modelo com a sua versão anterior. Considerando que o versionamento realiza esse procedimento, seria redundante a criação de um novo mecanismo de junção. Desse modo, a junção pode ser realizada pelo próprio mecanismo de versionamento. Contudo, para que a junção do mecanismo de versionamento seja usada, é necessário que o novo modelo-alvo seja interpretado como um modelo que existe no repositório. Para isso, é

necessário recuperar os dados de versionamento da última versão do modelo e inseri-los no novo modelo-alvo. Como é possível observar na Figura 4.14, este procedimento é equivalente a fazer o *check-out* do modelo, seguido do *check-in*. O procedimento, realizado para todos os elementos do modelo, é explicado na subseção 4.9.1.

4.9.1 Recuperação de Dados de Versionamento

Os elementos do modelo-alvo estão relacionados com os elementos do modelo-fonte em função do processo de transformação. Este relacionamento é registrado pelas **ligações de rastreabilidade** geradas durante a transformação. Além disso, a versão de um elemento está relacionada à sua versão anterior, caso não seja a primeira. Conforme é possível observar na Figura 4.15, as ligações de rastreabilidade definem relacionamentos no espaço, enquanto as ligações entre as diferentes versões determinam relacionamentos no tempo. Além disso, cada elemento possui um estereótipo contendo as informações de versionamento, que são usadas pelo mecanismo de versionamento para a localização, junção e versionamento dos elementos.

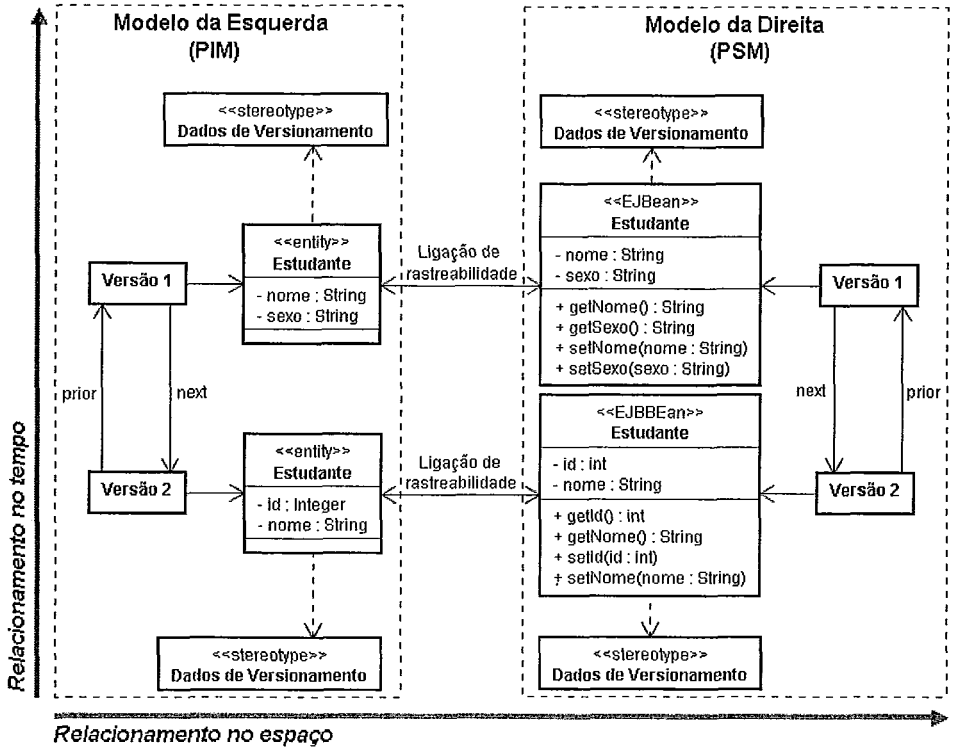

Figura 4.15: Relacionamentos no tempo e no espaço

No contexto do Odyssey-MEC, após a realização de uma transformação, o novo modelo-alvo não possui informações de versionamento. Usando o modelo da Figura

4.15 como exemplo, caso seja realizada uma transformação esquerda-direita, o segundo elemento *Estudante* do modelo da direita não terá o estereótipo contendo os dados de versionamento (Figura 4.16a). Assim, para que o mecanismo de versionamento interprete o novo elemento como sendo o mesmo que está no repositório, é necessário recuperar o estereótipo contendo os dados da versão anterior (Figura 4.16b). Para isso, é necessário localizar a versão anterior do elemento realizando uma busca no espaço e no tempo. A busca é iniciada no espaço a partir das ligações de rastreabilidade que foram geradas durante a transformação. No exemplo anterior, a ligação de rastreabilidade (Figura 4.16c) é usada para se identificar o elemento do modelo-fonte (Figura 4.16d) que foi usado para a criação do elemento do modelo-alvo.

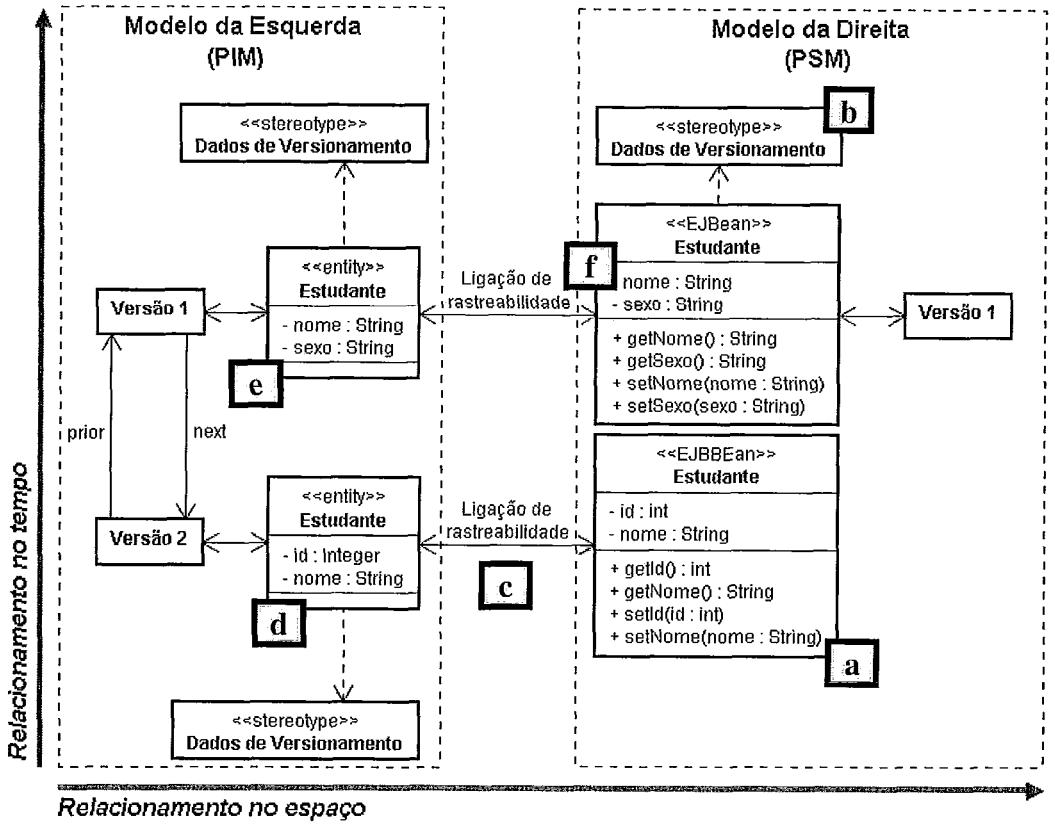

Figura 4.16: Procedimento de recuperação de dados de versionamento

Conforme descrito na seção 4.8 (Controle de Versão de Modelos), as informações sobre a versão de um elemento são mantidas pelo elemento *Version*. Além do número da versão, *Version* mantém referência para: o elemento UML que está sendo versionado (*EModelElement*), a versão anterior (*prior*) e a próxima versão (*next*). A partir dessa estrutura, é possível localizar a versão anterior do elemento do modelo-fonte usado na transformação (busca no tempo) (Figura 4.16e). A partir da versão anterior do elemento-

fonte e da identificação da transformação, a ligação de rastreabilidade criada na transformação anterior é localizada, assim como a versão anterior do elemento do modelo-alvo (nova busca no espaço) (Figura 4.16f). Finalmente, o estereótipo contendo os dados de versionamento (Figura 4.16b) pode ser recuperado e copiado para o novo elemento-alvo (Figura 4.16a). A partir desse momento, quando a máquina de sincronização fizer o *check-in* do novo modelo-alvo, o mecanismo de versionamento irá gerar uma nova versão para o modelo.

4.9.2 Localização de Elemento

Na subseção 4.9.1, foi descrito como o mecanismo de sincronização recupera os dados de versionamento de um elemento. A localização de um elemento no espaço é realizada com o auxílio das ligações de rastreabilidade geradas durante as transformações.

Durante o processo de transformação, é possível que a mesma transformação seja aplicada a elementos diferentes do modelo-fonte. Além disso, o mesmo elemento do modelo-fonte pode ser usado para a criação de vários elementos do modelo-alvo, e vários elementos do modelo-fonte podem estar relacionados ao mesmo elemento do modelo-alvo. Por exemplo, uma classe *Cliente* pode ser usada para gerar as classes *ClienteBean* e *ClientePK*, enquanto um método que tem como parâmetro a classe *Cliente* pode determinar a criação de um método no modelo-alvo, cujo parâmetro deve fazer referência a *ClienteBean*. Portanto, muitas ligações de rastreabilidade podem ter o mesmo identificador de transformação (Figura 4.17a) e fazer referência aos mesmos elementos do modelo-fonte (Figura 4.17b) ou do modelo-alvo (Figura 4.17c).

A possibilidade de vários rastros possuírem o mesmo identificador e fazer referência aos mesmos elementos torna necessária a combinação do **elemento-fonte** com o **identificador da transformação** para a localização de um **elemento-alvo**. Conforme pode ser observado na Figura 4.17, a combinação desses dois dados é suficiente para a localização da versão anterior do modelo-alvo. Por exemplo, a combinação de T1 com o elemento F1 possibilita a localização do elemento A1.

Vale a pena notar que cada transformação gera um conjunto de ligações de rastreabilidade separado. Além disso, uma transformação nunca é aplicada mais de uma vez no mesmo elemento. Portanto, não é possível, dentro de um mesmo conjunto de ligações de rastreabilidade, existir combinações *elemento_da_esquerda+transformação* e *elemento_da_direita+transformação* idênticos.

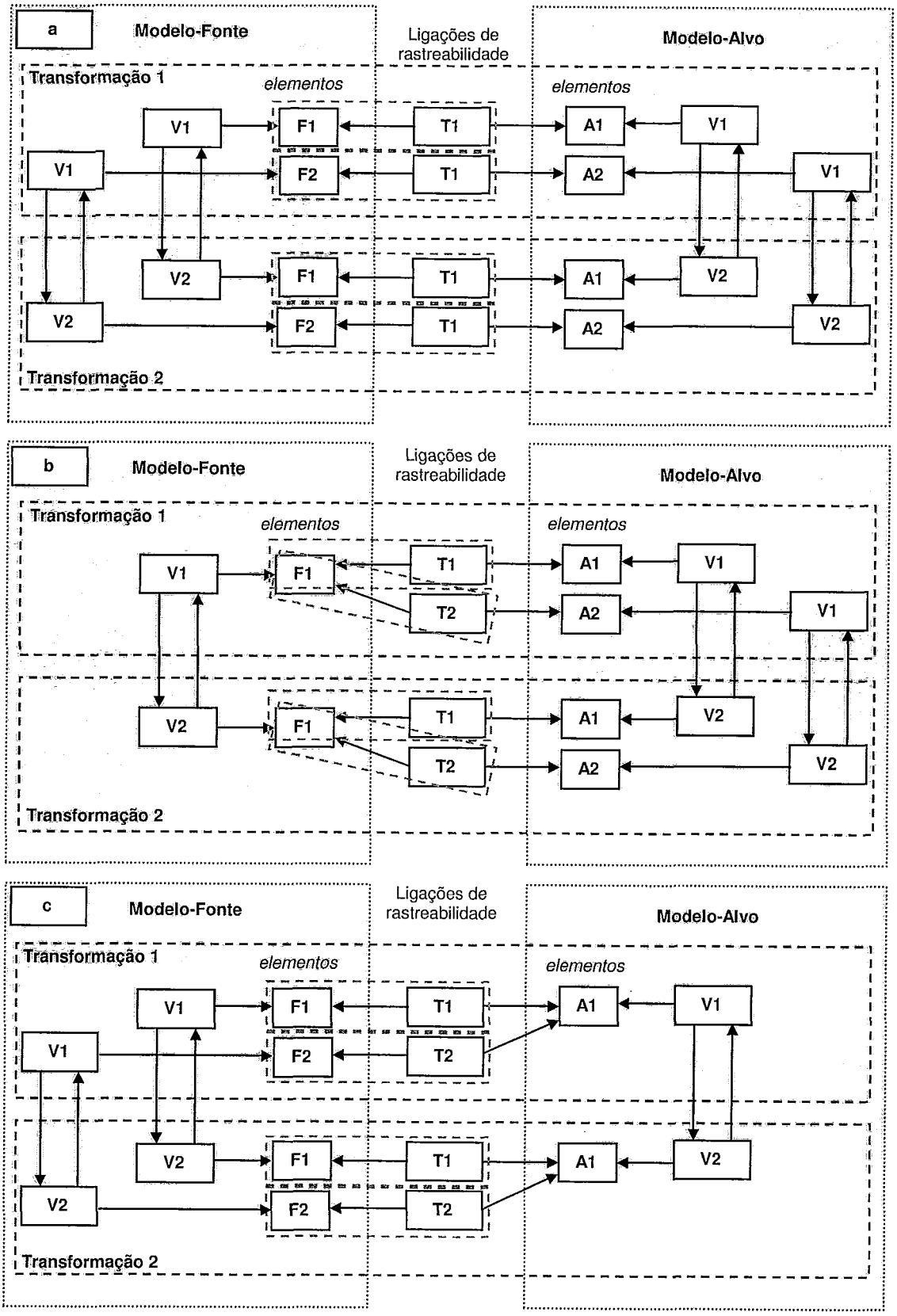

Figura 4.17: Ligações de rastreabilidade

4.9.3 Recuperação de Elementos Específicos do Modelo

Os procedimentos descritos nas subseções 4.9.1.e 4.9.2 possibilitam a recuperação de dados de versionamento de elementos do modelo-alvo que estão inter-relacionados com elementos do modelo-fonte. Contudo, é possível que a versão do modelo-alvo que está sendo recuperada tenha elementos que não estão relacionados com elementos do outro modelo. No caso de um PSM, por exemplo, esses elementos podem ter sido incluídos para tornar o modelo mais completo para a geração do código fonte. Portanto, como o modelo-fonte não possui elementos correspondentes, o novo modelo-alvo gerado pela transformação não terá os elementos existentes em sua versão atual. Como conseqüência, ao fazer a junção, o componente de versionamento excluiria esses elementos da nova versão do modelo. Para evitar esse problema, todos os elementos que existem apenas na versão atual do modelo-alvo são copiados para o novo modelo, incluindo as informações de versionamento. Contudo, é importante ressaltar que este procedimento não é realizado para os elementos que não deverão mais existir na nova versão em função dos elementos correspondentes do modelo-fonte terem sido excluídos.

4.9.4 Propagação da Sincronização

O Odyssey-MEC foi elaborado com a finalidade de transformar e sincronizar diferentes modelos computacionais de um projeto MDD. Considerando que a quantidade de modelos pode variar, é possível imaginar a relação entre os modelos como um grafo, conforme a Figura 4.18. Para que todos esses modelos permaneçam consistentes, é necessário propagar o processo de sincronização.

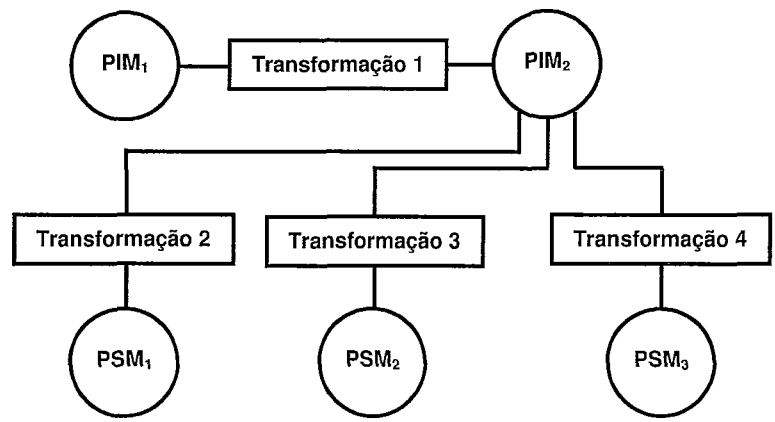

Figura 4.18: Exemplo de grafo de propagação de sincronização

Uma das tarefas realizadas pelo mecanismo de sincronização é a execução de transformação para gerar um novo modelo-alvo. Contudo, a direção da transformação pode variar em função do modelo que desencadeou o processo de sincronização. Por exemplo, se um usuário alterar o PIM_1 e fizer o *check-in*, transformações esquerda-direita serão executadas ao longo da sincronização, de modo que o PIM_2 seja sincronizado com o PIM_1 e os PSMs sejam sincronizados com o PIM_2. Contudo, se o modelo de origem for o PIM_2, uma transformação direita-esquerda será executada durante a sincronização desse modelo com o PIM_1, enquanto transformações esquerda-direita serão executadas para sincronizar os demais modelos (PSM_1, PSM_2 PSM_3) com o PIM_2. Finalmente, se o modelo de origem for o PSM_1, uma transformação direita-esquerda será executada durante a sincronização desse modelo com o PIM_2. Em seguida, outra transformação direita-esquerda será realizada para a sincronização do PIM_2 com o PIM_1. No final, transformações esquerda-direita serão executadas para sincronizar o PIM_2 com o PSM_2 e PSM_3.

4.10 Compartilhamento de Modelos

O desenvolvimento de sistemas de software grandes e complexos pode envolver vários profissionais com papéis específicos. Além disso, existe a possibilidade dos profissionais usarem ferramentas CASE diferentes, particularmente no caso em que a equipe de desenvolvimento é constituída por pessoas que estão distribuídas geograficamente. Quando este cenário é aplicado ao MDD com uma abordagem elaboracionista, isto significa que PIM e PSM podem ser manipulados a partir de ferramentas CASE distintas.

Para que qualquer ferramenta CASE possa ser utilizada para a criação de modelos, é necessário que a abordagem utilize um meio de compartilhamento que seja independente. A solução adotada é o emprego do XMI 2.1 (OMG, 2005b). Desse modo, qualquer ferramenta que exporte e importe modelos usando este padrão pode ser usada para a criação e manipulação de modelos. Contudo, é importante ressaltar que essas ferramentas devem ter condições de trabalhar com perfis UML e preservem as marcações aplicadas ao modelo pelo Odyssey-MEC ou seus componentes.

4.11 Integridade dos Modelos

O controle da evolução de modelos no contexto do MDD depende de processos que possibilitam a transformação, sincronização e versionamento dos modelos. Contudo, para que os modelos estejam sempre consistentes, é necessário que todo o processo seja executado com sucesso. Para isso, duas abordagens podem ser usadas. Na primeira, todas as alterações são aceitas e quaisquer problemas encontrados, como, por exemplo, alterações conflitantes, são apresentados em um relatório. Na segunda abordagem, as operações devem ser realizadas sem que ocorram conflitos. Nesta abordagem é necessário voltar o estado dos modelos à situação anterior ao início do processamento, caso qualquer uma das operações falhem. No caso do Odyssey-MEC, foi optado pela segunda abordagem, por ser mais conservadora.

Para manter a integridade dos modelos, as tarefas de transformação, sincronização e versionamento são realizadas dentro do contexto de transações aninhadas. Quando o desenvolvedor faz o *check-in* de um modelo, é iniciada uma transação para controlar o versionamento do modelo. Após o versionamento, uma nova transação é iniciada para fazer a sincronização de modelos. Após a sincronização, o *check-in* realizado pelo mecanismo de sincronização para versionar o novo modelo alvo inicia uma nova transação. Quando o último processo é encerrado, a respectiva transação é finalizada. Contudo, a falha de qualquer transação implica no cancelamento das alterações e os modelos voltam para o estado inicial.

4.12 Considerações Finais

Este capítulo descreveu o Odyssey-MEC, uma abordagem para o controle de evolução de modelos computacionais ao longo do tempo e no espaço. A forma como os requisitos identificados são atendidos é descrita a seguir.

- *Utilização de linguagens e formatos padronizados pela OMG*: A abordagem utiliza como padrão a UML como linguagem de modelagem. Desse modo, pode ser aplicada por técnicas de Desenvolvimento Dirigido por Modelos que usam o MDA como *framework*. Além disso, o XMI 2.1 é usado como referência para a persistência, importação e exportação de modelos. Contudo, a abordagem proposta nesta dissertação não usa o QVT (OMG, 2008) para a especificação de transformações. O motivo para esta decisão foi a indisponibilidade da versão final do QVT até o momento de definição da abordagem.

- *Uso de qualquer ferramenta CASE*: A abordagem permite a utilização de qualquer ferramenta CASE que exporte e importe modelos usando o formato do XMI 2.1.

- *Transformação automática de modelos*: O Odyssey-MEC realiza transformações automáticas de modelos, baseando-se em marcações e mapas de transformação, conforme definido para o Odyssey-MDA (MAIA, 2006).

- *Transformação bidirecional de modelos*: A abordagem realiza a transformação bidirecional de modelos. A direção da transformação é determinada pela configuração inicial do projeto e pelo modelo que está sendo recebido pelo Odyssey-MEC. Se o modelo estiver configurado como o da esquerda de uma transformação, a direção será da esquerda para a direita, e vice-versa.

- *Transformação independente de desenvolvedor*: As transformações são realizadas no lado servidor, sendo independentes da iniciativa do desenvolvedor. O processo é iniciado, automaticamente, quando o usuário faz o *check-in* do modelo. Esse procedimento garante a realização da transformação e auxilia da manutenção da consistência dos modelos no espaço.

- *Sincronização automática de modelos:* Os modelos são sincronizados automaticamente pela abordagem, tornando desnecessário qualquer trabalho manual, evitando, desse modo, os possíveis erros que poderiam ser cometidos pelos desenvolvedores.

- *Sincronização bidirecional de modelos*: Assim como a transformação, a abordagem realiza a sincronização dos modelos nas duas direções. A direção da sincronização é determinada pela configuração da transformação do projeto e pelo modelo que está sendo disponibilizado para o Odyssey-MEC.

- *Sincronização independente de desenvolvedor*: O processo de sincronização é disparado quando o desenvolvedor disponibiliza um modelo para o Odyssey-MEC, a partir da operação de *check-in*. Portanto, não depende da iniciativa do desenvolvedor.

- *Geração de ligações de rastreabilidade durante a transformação*: As ligações de rastreabilidade são geradas durante as transformações. No Odyssey-MEC,

essas ligações estão sendo usadas para auxiliar a sincronização de modelos. Contudo, também estão disponíveis para serem usadas com outras finalidades.

- *Controle de versões de modelos*: a abordagem proposta gera novas versões dos modelos sempre que são modificados, mantendo sob controle a evolução dos modelos ao longo do tempo, conforme especificado para o Odyssey-VCS (OLIVEIRA, 2005).

- *Suporte ao desenvolvimento distribuído*: O uso de uma arquitetura cliente-servidor possibilita que os desenvolvedores possam atuar no projeto de qualquer lugar, mesmo estando geograficamente distribuídos.

Tabela 4.4: Quadro comparativo das abordagens

Requisitos	Abordagens									
	1	2	3	4	5	6	7	8	9	10
	C-SAW	Xiong et al	SMoVER	Marcus Alanen	Sriplakich et al	Odyssey-VCS	Odyssey-MDA	Together	Enterprice Architect	Odyssey-MEC
Utilização de linguagens e formatos padrões da OMG	±	±	±	✓	✓	✓	±	✓	±	±
Uso de qualquer ferramenta CASE	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓
Transformação automática de modelos	✓	✓	✗	✗	✗	✗	✓	✓	✓	✓
Transformação bidirecional de modelos	✗	✓	✗	✗	✗	✗	✓	±	✗	✓
Transformação independente de desenvolvedor	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Sincronização automática de modelos	✗	✓	✗	✗	✗	✗	✓	✗	✓	✓
Sincronização bidirecional de modelos (*round-trip*)	✗	✓	✗	✗	✗	✗	✓	✗	✗	✓
Sincronização independente de desenvolvedor	✗	✓	✗	✗	✗	✗	✓	✗	✓	✓
Geração de ligações de rastreabilidade	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓
Controle de versões de modelos	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓
Suporte ao desenvolvimento distribuído	✗	✗	✓	✓	✓	✓	✗	✗	✓	✓

A comparação do Odyssey-MEC com as abordagens descritas no capítulo 3 é apresentada na Tabela 4.4. Além de ser a única que permite o controle da evolução de modelos no tempo e no espaço, é possível destacar a realização de transformação e sincronização independente da iniciativa do desenvolvedor e a geração de ligações de rastreabilidade.

Para que o Odyssey-MEC possa controlar a evolução de modelos, duas abordagens foram reutilizadas: Odyssey-MDA, para a realização de transformações, e o Odyssey-VCS, para o controle de versão dos modelos. A relação dessas abordagens, juntamente com o Odyssey-MEC, com as dimensões tempo e espaço é apresentada na Tabela 4.5. Como é possível observar, o Odyssey-VCS trata modelos isoladamente (mesmo espaço) ao longo do tempo, enquanto o Odyssey-MDA trata modelos que estão em espaços diferentes, mas que estão formalmente inter-relacionados. Com o auxílio desses componentes, o Odyssey-MEC controla os modelos nas duas dimensões.

Tabela 4.5: Odyssey-MEC e seus componentes em relação ao tempo e ao espaço

Espaço / Tempo	**Mesmo**	**Diferente**
Mesmo	Modelagem UML convencional (modelos isolados não versionados)	MDA convencional (modelos formalmente inter-relacionados e não versionados) **(Odyssey-MDA)**
Diferente	Modelagem UML com Versionamento **(Odyssey-VCS)**	MDA Versionada **(Odyssey-MEC)**

No Capítulo 5, são apresentados alguns exemplos de aplicação da abordagem Odyssey-MEC, considerando os aspetos apresentados neste capítulo.

Capítulo 5 – Exemplos de Aplicação da Abordagem

5.1 Introdução

No capítulo anterior, foi apresentada a abordagem Odyssey-MEC. Neste capítulo, são apresentados alguns exemplos focados no processo de transformação, sincronização e versionamento dos modelos. O objetivo é demonstrar que a abordagem permite o controle da evolução de modelos inter-relacionados no tempo e no espaço. Esses exemplos estão baseados nos modelos apresentados na Figura 5.1. O modelo de casos de uso e o modelo de classes representam um sistema para controle de reservas de um hotel. Os processamentos relacionados com os casos de uso "Registrar reserva" e "Cancelar reserva" são realizados pela classe *ControladorReserva*. Nos exemplos, serão considerados *PSMs* do sistema para as plataformas *JEE* e *C#* com *.NET*. Considerando que o processo de sincronização é realizado partir do momento que existe a primeira versão dos modelos-alvo, os exemplos serão apresentados a partir deste cenário.

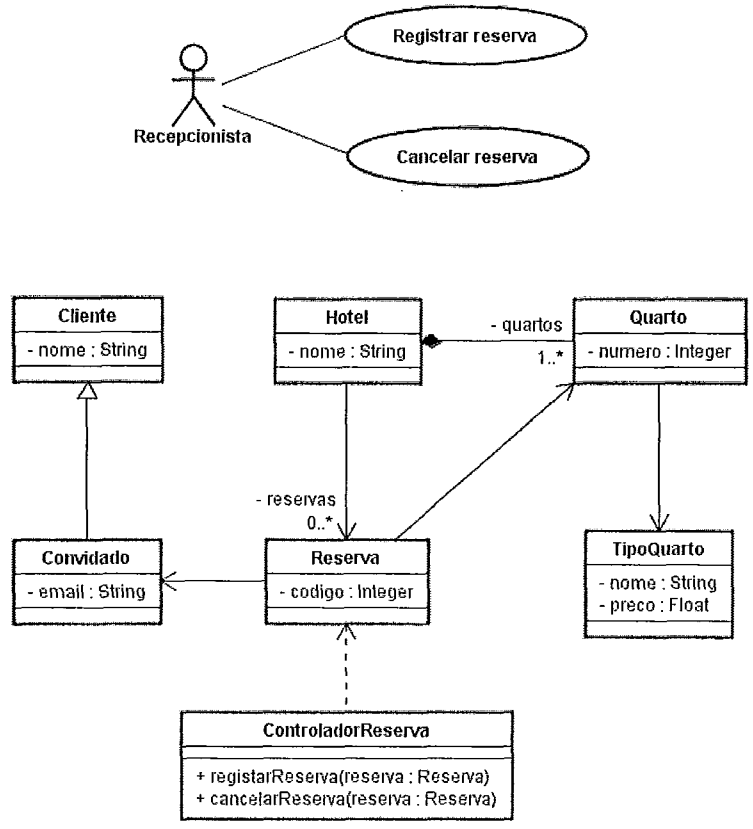

Figura 5.1: Modelo de casos de uso de classes (PIM) usado nos exemplos

Os exemplos que serão abordados são: (1) inclusão de um elemento transformável no PIM; (2) alteração de um elemento transformável do PIM; (3) exclusão de um elemento transformável do PIM; (4) inclusão de um elemento transformável no PSM; (5) alteração de um elemento transformável do PSM; (6) exclusão de um elemento transformável do PSM; (7) inclusão de um elemento não transformável no PSM; (8) alteração de um elemento não transformável do PSM; (9) exclusão de um elemento não transformável do PSM; (10) conflito de versões do mesmo modelo; e (11) conflito de versões de modelos diferentes.

5.2 Inclusão de um Elemento Transformável no PIM

A inclusão de um elemento transformável no PIM implica na criação de uma nova versão do modelo, assim como a inclusão de novos elementos nos PSMs do projeto e na criação de novas versões desses modelos. Por exemplo, a inclusão do atributo *telefone* na classe *Cliente*, faz com que uma nova versão do PIM seja criada. Nesse caso, o atributo *nome* e a classe *Cliente* passam para a versão 2. Após o versionamento, o PIM é usado como modelo-fonte na transformação que irá criar um novo modelo-alvo PSM-JEE. Comparando o novo modelo-alvo com a versão anterior, podem ser identificadas as seguintes operações no PSM-JEE: (1) inclusão do atributo *nome* na classe *ClienteBean*; e (2) inclusão dos métodos *getNome* e *setNome* na classe *ClienteBean*. Portanto, esses elementos são novos no PSM-JEE.

Para fazer o versionamento do novo PSM-JEE, a máquina de sincronização deve recuperar as informações de versionamento da última versão do novo modelo-alvo. Para isso, a máquina de sincronização realiza o *check-out* da versão atual do PSM-JEE e realiza o processo de localização dos elementos e recuperação das informações de versionamento (Capítulo 4). Para cada elemento do modelo-fonte, as ligações de rastreabilidade geradas na última transformação e na anterior são usadas para a localização da versão anterior do elemento. Como os elementos *nome*, *getNome* e *setNome* são novos, não existem os dados de versionamento desses elementos.

O novo PSM-JEE possui apenas os elementos provenientes da transformação. Contudo, a versão anterior do modelo pode possuir elementos específicos, que foram incluídos manualmente. Esses elementos e os respectivos dados de versionamento são incluídos no novo modelo, de modo que o modelo permaneça com os mesmos elementos.

Após o término da recuperação dos dados de versionamento e a recuperação de elementos específicos do modelo, é executado o *check-in* do PSM-JEE. Assim, é criada a segunda versão desse modelo. Nesse caso, os elementos *nome*, *getNome* e *setNome* recebem a versão 2. Como esses elementos fazem parte da classe *ClienteBean*, esse elemento também passa para a versão 2.

Após a sincronização do PSM-JEE, o mesmo processo se repete para o PSM-CNET. Nesse caso, a transformação implica na criação do atributo *nome* e dos métodos *getNome* e *setNome* na classe *Cliente* de PSM-CNET.

5.3 Alteração de um Elemento Transformável no PIM

A alteração de um elemento transformável no PIM implica na criação de uma nova versão do modelo e na alteração e versionamento dos elementos correspondentes nos PSMs do projeto. Por exemplo, a alteração do atributo *nome* da classe *HotelBean* para *razaoSocial* faz com que o atributo e a classe recebam a versão 3 (considerando que a versão 2 do modelo foi criada após a inclusão do atributo *telefone* na classe *Cliente* do PIM). Logo em seguida, o PIM é usado como modelo-fonte na transformação que gera o novo modelo PSM-JEE. Comparando-se o novo modelo com a versão atual, é possível identificar as seguintes modificações na classe *HotelBean*: (1) alteração do atributo *nome* para *razaoSocial*; (2) alteração do método *getNome* para *getRazaoSocial*; e (3) alteração do método *setNome* para *setRazaoSocial*.

Após a recuperação da versão anterior do PSM-JEE, é executado o processo de localização dos elementos da versão anterior do PSM-JEE a partir das ligações de rastreabilidade e das ligações da versão atual do PIM com a versão anterior do mesmo elemento (Capítulo 4). Neste caso, como a transformação é correspondente à alteração de elementos do PSM-JEE, os dados de versionamento de todos os elementos são recuperados. Quando o novo PSM-JEE passa pelo processo de versionamento, os elementos alterados passam para a versão 3, assim como a classe *HotelBean*.

Assim como no caso da inclusão de um elemento, o novo PSM-JEE possui apenas os elementos provenientes da transformação. Contudo, a versão anterior do modelo pode possuir elementos que foram incluídos manualmente. Esses elementos e os respectivos dados de versionamento são incluídos no novo modelo, de modo que o modelo permaneça com os mesmos elementos após o versionamento.

Após a sincronização do PSM-JEE, o mesmo processo se repete para o PSM-CNET. Nesse caso, a transformação implica na alteração do atributo *nome* e dos

métodos *getNome* e *setNome* na classe *Hotel* de PSM-CNET para *razaoSocial*, *getRazaoSocial* e *setRazaoSocial*, respectivamente.

5.4 Exclusão de um Elemento Transformável do PIM

A exclusão de um elemento transformável do PIM implica na exclusão dos elementos correspondentes dos PSMs do projeto e na criação de novas versões dos modelos. Por exemplo, a exclusão do atributo *email* da classe *Convidado* faz com que uma nova versão PIM seja criada. Nesse caso, a classe *Convidado* recebe a versão 4.

Após o versionamento, o PIM é usado como modelo-fonte para a criação do modelo-alvo PSM-JEE a partir do processo de transformação. Nesse caso, a classe *ConvidadoBean* do novo modelo PSM-JEE não possuirá o atributo *email*, nem os métodos *getEmail* e *setEmail*.

Durante a recuperação dos dados de versionamento do PSM-JEE, como a quantidade de elementos do novo modelo é inferior à versão atual, todos os dados de versionamento são recuperados. Além disso, os elementos específicos do modelo, incluídos manualmente, também são recuperados, de modo que o modelo permaneça com os mesmos elementos que a versão anterior.

Quando o PSM-JEE é versionado, o mecanismo de junção identifica que os elementos *email*, *getEmail* e *setEmail* não existem no novo PSM-JEE. Esses elementos são considerados como excluídos e, como conseqüência, o modelo recebe a versão 4.

Após a sincronização do PSM-JEE, o mesmo processo se repete para o PSM-CNET. Nesse caso, a transformação implica na exclusão do atributo *email* e dos métodos *getEmail* e *setEmail* da classe *ConvidadoBean* de PSM-CNET.

5.5 Inclusão de um Elemento Transformável no PSM

A inclusão de um elemento transformável no PSM implica na mesma operação para o PIM e para os outros PSMs do projeto, além de uma nova versão para cada modelo. Por exemplo, a inclusão da classe *AtendenteBean* no PSM-JEE, com o estereótipo *EntityBean*, faz com que uma nova versão do modelo seja criada (versão 5). Em seguida, o PSM-JEE é usado como modelo-fonte em uma transformação que irá gerar o novo PIM, que terá uma classe *Atendente*, com o estereótipo *Entity*.

Para fazer o versionamento do novo PIM, os dados da versão anterior do modelo são recuperados. Como a classe *Atendente* é nova, não existem dados de versionamento

para esse elemento. Durante o versionamento do PIM, a classe *Atendente* recebe a versão 5, assim como o modelo.

A criação da nova classe *Atendente* no PIM faz com que o PSM-CNET tenha de ser atualizado. Desse modo, é realizada uma transformação esquerda-direita, em que o PIM é o modelo-fonte para a criação do novo modelo PSM-CNET, contendo a classe *Atendente*. Assim como ocorreu com a nova versão do PIM, os dados de versionamento dos elementos previamente existentes são recuperados. Ao ser versionado, a classe *Atendente* do PSM-CNET recebe a versão 5, assim como o modelo.

5.6 Alteração de um Elemento Transformável do PSM

A alteração de um elemento transformável do PSM implica na mesma operação para o PIM e para os outros PSMs do projeto, além de uma nova versão para cada modelo. Por exemplo, a alteração do nome da classe *AtendenteBean* do PSM-JEE para *GerenteBean*, faz com que seja criada uma nova versão para a classe e para o respectivo modelo (versão 6). Em seguida, o PSM-JEE é usado como modelo-fonte em uma transformação direita-esquerda, que irá gerar o novo PIM com a classe *Gerente*.

Para fazer o versionamento do novo PIM, os dados da versão anterior do modelo são recuperados a partir das ligações de rastreabilidade e das informações de versionamento do modelo-fonte (Capítulo 4). Como a classe *Gerente* do PSM-JEE está associada com a classe *Atendente* da versão anterior do mesmo modelo, os dados de versionamento são recuperados e colocados na classe *Gerente*. Durante o processo de versionamento, a classe *Gerente* recebe a versão 6, assim como o PIM.

A alteração da classe *Atendente* do PIM faz com que o PSM-CNET tenha de ser atualizado. Desse modo, é realizada uma transformação esquerda-direita, em que o PIM é o modelo-fonte para a criação do novo modelo PSM-CNET com a classe *Gerente*. Assim como ocorreu com a nova versão do PIM, os dados de versionamento dos elementos previamente existentes são recuperados. Ao ser versionado, a classe *Gerente* do PSM-CNET recebe a versão 6, assim como o modelo.

5.7 Exclusão de um Elemento Transformável do PSM

A exclusão de um elemento transformável do PSM implica na mesma operação para o PIM e para os outros PSMs do projeto, além de uma nova versão para cada modelo. Por exemplo, a exclusão da classe *GerenteBean* do PSM-JEE faz com que uma nova versão do modelo seja criada (versão 7). Em seguida, o PSM-JEE é usado como

modelo-fonte em uma transformação que irá gerar o novo PIM, que não terá a classe *Gerente*.

Para fazer o versionamento do novo PIM, os dados da versão anterior do modelo são recuperados. Contudo, como a classe *Gerente* não existe no PIM novo, os dados de versionamento referente a esse elemento não são recuperados. Durante o versionamento do PIM, a classe *Gerente* é considerada como excluída do modelo, que recebe a versão 7.

A não geração da classe *Gerente* do PIM faz com que o PSM-CNET tenha de ser atualizado. Desse modo, é realizada uma transformação esquerda-direita, em que o PIM é o modelo-fonte para a criação do novo modelo PSM-CNET. Assim como o PIM, esse modelo não contém a classe *Gerente*. Os dados de versionamento dos elementos previamente existentes são recuperados, menos da classe *Gerente*. Ao ser versionado, a classe *Gerente* é considerada como excluída do PSM-CNET e, assim como o PIM, recebe uma nova versão (número 7).

5.8 Inclusão de um Elemento não Transformável no PSM

Nos exemplos apresentados até o momento, as alterações realizadas em um modelo implicaram nas mesmas operações nos modelos-alvo. Contudo, é possível que um modelo receba modificações que não devem ser propagadas para outros modelos. Por exemplo, a inclusão do método privado *localizarPorCodigo* na a classe *ControladorReservaBean* do PSM-JEE não deve ser propagada para o PIM. Contudo, o modelo ainda deve ser versionado. Desse modo, durante o processo de versionamento, o método e a classe recebem a versão 8. Em seguida, o PSM-JEE é usado como modelo-fonte para a transformação que gera o PIM. Como a inclusão do método *localizarPorCodigo* não é propagada para o PIM, o novo modelo possui os mesmos elementos que sua versão anterior. Assim, os dados de versionamento de todos os elementos do PIM são recuperados.

Ao tentar criar uma nova versão do PIM, o mecanismo de versionamento não encontra alterações, fazendo com que o modelo permaneça na versão 7. Como o PIM não foi modificado, a transformação PIM→PSM-CNET não é executada. Como conseqüência, o PSM-CNET também permanece na versão 7.

5.9 Alteração de um Elemento não Transformável do PSM

Assim como ocorre a inclusão de um elemento não transformável em um modelo, o mesmo ocorre em relação à alteração desse tipo de elemento. Por exemplo, a alteração do método privado *localizarPorCodigo* da a classe *ControladorReservaBean* do PSM-JEE para *localizarPorId* não deve ser propagada para o PIM. Contudo, o modelo ainda deve ser versionado. Desse modo, durante o processo de versionamento, o método e a classe recebem a versão 9, assim como o PSM-JEE. Em seguida, o PSM-JEE é usado como modelo-fonte para a transformação direita-esquerda que gera o PIM. Como a alteração do método *localizarPorCodigo* não é propagada para o PIM, o novo modelo possui exatamente os mesmos elementos que sua versão anterior. Assim, os dados de versionamento de todos os elementos do PIM são recuperados.

Ao tentar criar uma nova versão do PIM, o mecanismo de versionamento não encontra alterações, fazendo com que o modelo permaneça na versão 7. Como o PIM não foi modificado, a transformação PIM→PSM-CNET não é executada. Como conseqüência, o PSM-CNET também permanece na versão 7.

5.10 Exclusão de um Elemento não Transformável do PSM

Assim como ocorre a inclusão e a alteração de um elemento não transformável, o mesmo ocorre em relação à exclusão. Por exemplo, a exclusão do método privado *localizarPorId* da a classe *ControladorReservaBean* do PSM-JEE não deve ser propagada para o PIM. Contudo, o modelo ainda deve ser versionado. Desse modo, durante o processo de versionamento, a classe *ControladorReservaBean* recebe a versão 10, assim como o PSM-JEE. Em seguida, o modelo é usado como entrada da transformação que gera o PIM. Como a exclusão do método não é propagada para o PIM, o novo modelo possui exatamente os mesmos elementos que sua versão anterior. Assim, os dados de versionamento de todos os elementos do PIM são recuperados.

Ao tentar criar uma nova versão do PIM, o mecanismo de versionamento não encontra alterações, fazendo com que o modelo permaneça na versão 7. Como o PIM não foi modificado, a transformação PIM→PSM-CNET não é executada. Como conseqüência, o PSM-CNET também permanece na versão 7.

5.11 Conflito a partir da Alteração do Mesmo Modelo

Um conflito ocorre quando dois desenvolvedores realizam uma operação de inclusão, alteração ou exclusão de um mesmo elemento do modelo. O tratamento de conflitos é realizado durante o processo de versionamento (Capítulo 4). Por exemplo, um conflito pode ocorrer, quando um desenvolvedor altera o nome da classe *Cliente* para *Hospede*, enquanto outro modifica o nome da mesma classe para *Usuario*. Nesse caso, como o conflito ocorre no momento do versionamento do modelo-fonte, toda a operação é cancelada. Desse modo, o processo de sincronização não é executado.

5.12 Conflito a partir de Alterações de Modelos Diferentes

Além do conflito provocado pela alteração do mesmo modelo, é possível que o mesmo problema aconteça quando modelos diferentes são modificados. Esse caso pode ocorrer, por exemplo, quando dois desenvolvedores, ao mesmo tempo, fazem o *check-out* do PIM e do PSM, respectivamente, e alteram elementos inter-relacionados. Por exemplo, o desenvolvedor João altera o atributo *nome* da classe *Hotel* do PIM para *nomeFantasia*. Após fazer o *check-in* desse modelo, os elementos correspondentes do PSM-JEE são alterados e recebem uma nova versão. Neste tempo, o desenvolvedor *Pedro* altera o nome do atributo correspondente no PSM-JEE para *razaoSocial*. Quando Pedro faz o *check-in* do modelo, a alteração realizada entre em conflito com a modificação decorrente da transformação. Assim como no caso de conflito decorrente de alterações realizadas no mesmo modelo, toda a operação é cancelada.

5.13 Considerações Finais

Este capítulo apresentou alguns exemplos sobre a aplicação da abordagem para o controle da evolução de modelos inter-relacionados ao longo do tempo e do espaço. Os exemplos consideraram a inclusão, alteração e exclusão de elementos. A realização dessas tarefas em elementos transformáveis do modelo-fonte implica, automaticamente, nas mesmas operações nos modelos-alvo. Contudo, quando os elementos não são transformáveis, o processo de sincronização não é realizado para esses elementos.

Outro aspecto a ser notado é a recuperação de elementos que existe apenas no modelo-alvo. Esses elementos são inseridos manualmente, o que normalmente é decorrente da necessidade de tornar o modelo mais completo, como, por exemplo, detalhar o PSM para a geração de código fonte. Logo após a transformação, esses

elementos não existem no novo modelo-alvo. Portanto, os elementos específicos do modelo devem ser recuperados da versão anterior.

Além dos exemplos de propagação e não propagação de alterações, foram apresentadas situações de conflitos decorrentes de alterações no mesmo modelo e em modelos diferentes. Em ambos os casos, toda a operação é cancelada, de modo que os modelos do projeto permaneçam consistentes entre si.

No Capítulo 6, é apresentado o protótipo do Odyssey-MEC e alguns exemplos que demonstram o seu uso, levando em consideração alguns exemplos apresentados neste capítulo.

Capítulo 6 – Protótipo do Odysssey-MEC

6.1 Introdução

No Capítulo 4, foi apresentada a proposta do Odyssey-MEC, uma abordagem para o controle da evolução de modelos computacionais criados segundo o Desenvolvimento Dirigido por Modelos (BEYDEDA *et al.*, 2005), utilizando o *framework* MDA (OMG, 2003b). A finalidade é possibilitar que os modelos computacionais inter-relacionados de um projeto MDD possam evoluir de maneira consistente no tempo e no espaço. A evolução ao longo do tempo é controlada através do controle de versões dos modelos, enquanto a evolução no espaço é realizada mantendo os modelos sincronizados. Alguns exemplos de como a abordagem controla a evolução especial e temporal dos modelos foram apresentados no Capítulo 5.

Este capítulo apresenta o protótipo construído para apoiar a utilização da abordagem Odyssey-MEC (CORRÊA *et al.*, 2008a; CORRÊA *et al.*, 2008b). O protótipo foi implementado em Java (SUN, 2008), usando o EMF (BUDINSKY *et al.*, 2003) e a UML2 (ECLIPSE, 2008b) para a manipulação e persistência de modelos. Além disso, o *Model Transaction* (ECLIPSE, 2008c) foi usado como base para a criação do gerenciador de transações, e a arquitetura cliente-servidor foi implementada a partir de *Web Services* (BOOTH *et al.*, 2005). A implementação do protótipo foi realizada de modo a possibilitar o controle da evolução de modelos criados em qualquer ferramenta CASE capaz de importar e exportar modelos UML 2.x usando o XMI 2.1.

Este capítulo está organizado da seguinte maneira: a Seção 6.2 apresenta o contexto de utilização do protótipo. A Seção 6.3 descreve a implementação do protótipo. O uso e os exemplos relacionados ao protótipo são apresentados nas seções 6.4 e 6.5, respectivamente. Finalmente, as considerações finais do capítulo são apresentadas na Seção 6.6.

6.2 Contexto de Uso

Esta seção apresenta a utilização do Odyssey-MEC de forma independente de ferramenta CASE. Para isso, foi implementado um aplicativo cliente que realiza as operações comuns a um sistema de controle de versão, como *check-in*, *check-out*, listagem do conteúdo do repositório e importação e exportação de modelos. Além disso,

o cliente do Odyssey-MEC possibilita a marcação de modelos. Esse aplicativo foi criado com base no cliente do *Odyssey-VCS* (OLIVEIRA, 2005), de modo a minimizar o esforço de aprendizado das pessoas que já estejam familiarizadas com esta ferramenta.

Para que os modelos possam ser transformados e sincronizados independentemente da iniciativa dos engenheiros de software, é necessário informar os modelos do projeto e associar cada modelo da esquerda com o respectivo modelo da direita. Para isso, foi criado um gerenciador de projetos.

A subseção 6.2.1 apresenta o Eclipse como ferramenta para a criação de modelos cuja evolução pode ser controlada pelo Odyssey-MEC. A interoperabilidade é realizada a partir da importação e exportação de modelos usando o XMI 2.1.

6.2.1 Eclipse como Ferramenta de Modelagem

O Eclipse (Figura 6.1) (ECLIPSE, 2008a) é um ambiente para implementação de programas usado principalmente para a linguagem Java. Contudo, recursos vêm sendo desenvolvidos para possibilitar a criação de modelos no próprio ambiente, sendo o EMF (BUDINSKY *et al.*, 2003) a base para a criação da infra-estrutura. A partir do EMF, é possível, por exemplo, criar um editor de modelos. Além disso, *frameworks,* como o GEF (ECLIPSE, 2008d), possibilitam a produção de *plugins* para a edição de modelos a partir de diagramas.

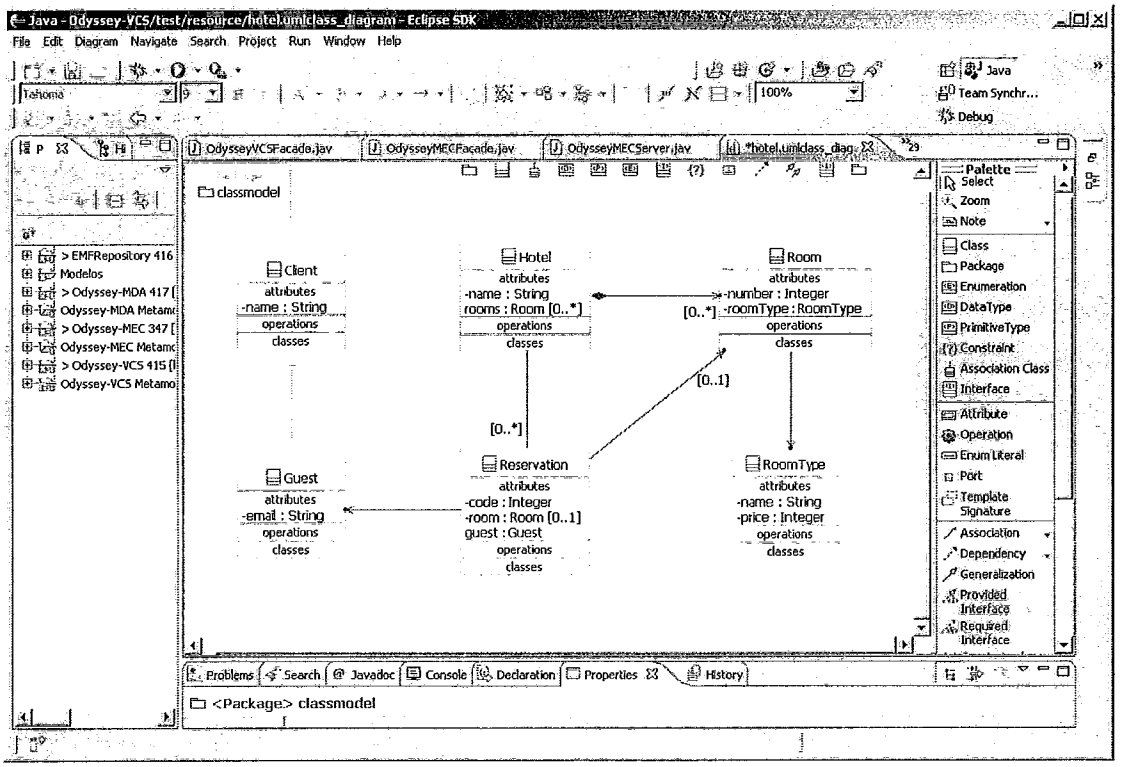

Figura 6.1: Eclipse como ambiente de modelagem

O Eclipse está sendo usado como exemplo em função de gerar modelos UML e fazer a exportação de acordo com o formato XMI 2.1. Para manter a evolução de um modelo criado no Eclipse junto com os outros modelos de um projeto MDD, é necessário apenas exportar o modelo do Eclipse, importar o modelo para o aplicativo cliente do Odyssey-MEC e fazer o *check-in*.

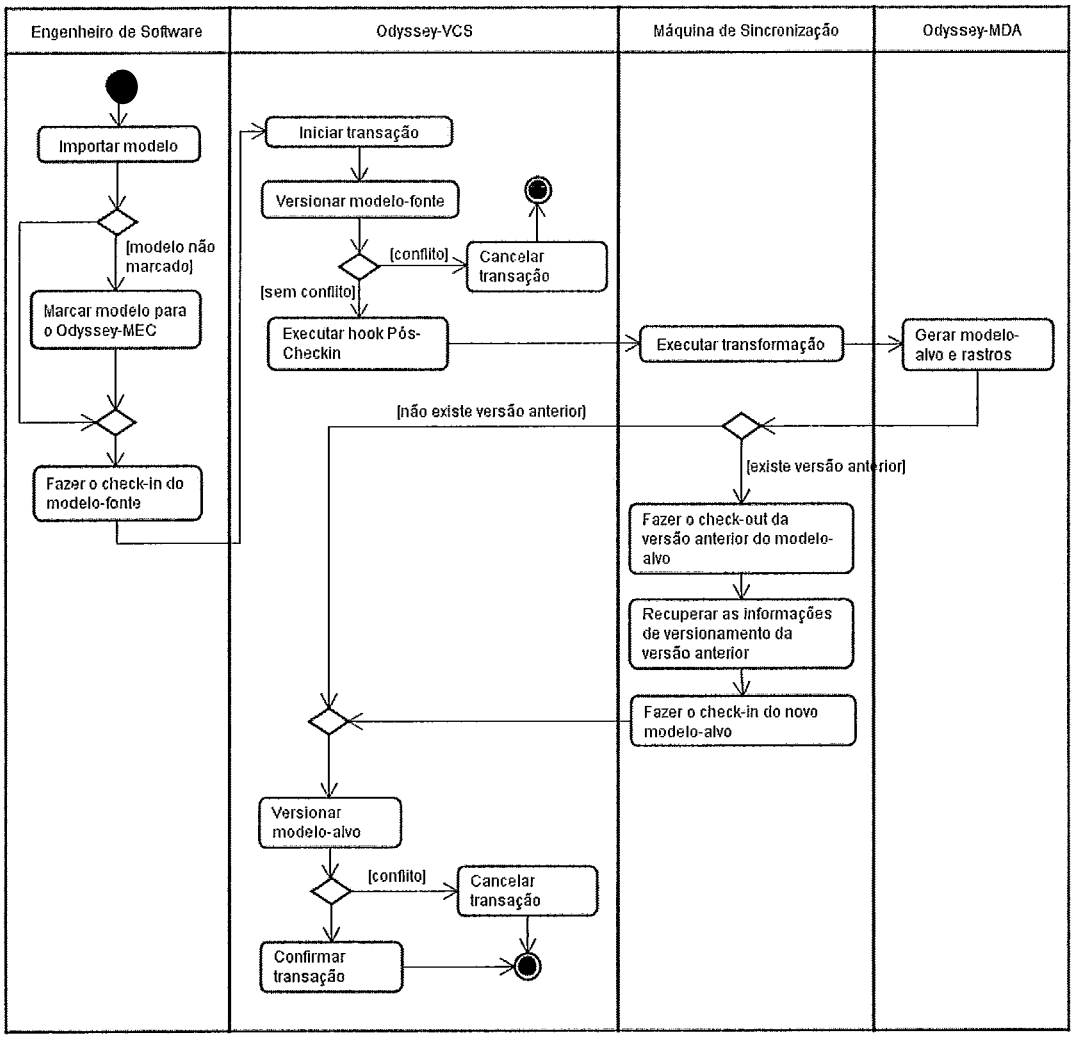

Figura 6.2: Processo realizado para o versionamento, transformação e sincronização de modelos

6.2.2 Utilização do Odyssey-MEC

O principal objetivo do Odyssey-MEC é executar a transformação e sincronização automática de modelos após a realização da operação de *check-in*. O processo relacionado com a realização dessa tarefa e a participação do engenheiro de software e dos componentes do Odyssey-MEC são apresentados na Figura 6.2. Como pode ser

observado, o engenheiro de software deve importar o modelo para o ambiente do Odyssey-MEC. Se necessário, o modelo deve receber as marcações a serem usadas pelo componente de transformação Odyssey-MDA (MAIA, 2006). O engenheiro de software deve fazer o *check-in* do modelo para iniciar o processo de versionamento, transformação e sincronização de modelos. O *check-in* faz com que uma transação seja iniciada. Em seguida, o componente de versionamento Odyssey-VCS (OLIVEIRA *et al.*, 2005) tenta criar uma nova versão do modelo (modelo-fonte). Se houver algum conflito, a transação é cancelada. Caso contrário, o *hook pós-checkin* aciona a máquina de sincronização, que executa o Odyssey-MDA para fazer a transformação necessária para gerar o modelo-alvo. Se não existir uma versão anterior do modelo-alvo no respectivo repositório, a máquina de sincronização usa o Odyssey-VCS para fazer o *check-in* do modelo. Se existir uma versão anterior, antes de fazer o *check-in*, a máquina de sincronização faz o *check-out* da versão anterior do modelo-alvo para recuperar as informações de versionamento. No final do processo, caso não ocorram problemas, como conflito com o modelo-alvo, por exemplo, a transação é confirmada. No caso de ocorrer algum problema, a transação é cancelada. Os detalhes de implementação que possibilitam a realização desse processo são descritos na seção 6.3.

6.3 Detalhamento da Implementação

O Odyssey-MEC e seus componentes utilizam o EMF (*Eclipse Modeling Framework*) (BUDINSKY *et al.*, 2003) como infra-estrutura para a manipulação, acesso e persistência de modelos. O EMF possibilita o uso de modelos dentro do ambiente do Eclipse (ECLIPSE, 2008a). Contudo, o EMF pode ser usado para a criação de aplicações independentes. Além disso, o EMF permite a manipulação de qualquer modelo cujo meta-modelo seja especificado a partir do *Ecore* (BUDINSKY *et al.*, 2003). Este meta-metamodelo é a solução alternativa ao MOF, criada para o Eclipse. Apesar de não ser um padrão da OMG, a interoperabilidade de modelos não é comprometida, pois o EMF faz a importação e exportação de modelos usando como padrão o XMI 2.1 (OMG, 2005b).

Para a manipulação de modelos UML 2.0, é utilizado o UML2 (ECLIPSE, 2006), uma implementação do meta-modelo UML baseada no EMF (ECLIPSE, 2006). A implementação obedece rigorosamente à especificação definida pela OMG.

A visão geral da arquitetura do Odyssey-MEC é apresentada na Figura 6.3. O gerenciador de projetos MDD (1) permite ao gerente de projeto criar um projeto MDD e

definir os respectivos modelos. Essas informações são mantidas no repositório de projetos (2). O aplicativo cliente do Odyssey-MEC (3) disponibiliza para o engenheiro de software as operações de *check-in* e *check-out*, além de permitir a marcação de modelos a partir do componente *ModelMarker* (MAIA, 2006). O servidor é constituído dos seguintes componentes: Odyssey-VCS (MURTA *et al.*, 2008), Odyssey-MDA (MAIA, 2006), máquina de sincronização e gerenciador de transação. O Odyssey-VCS (4) é usado para o versionamento dos modelos, enquanto o Odyssey-MDA (5) é usado para a transformação. A máquina de sincronização (6) realiza a sincronização dos modelos inter-relacionados. Finalmente, o gerenciador de transação (7) tem como finalidade fazer com que o versionamento, a transformação e a sincronização sejam realizadas dentro do contexto de uma transação, de modo que, se alguma dessas tarefas falhar, o projeto permaneça no estado anterior à transação.

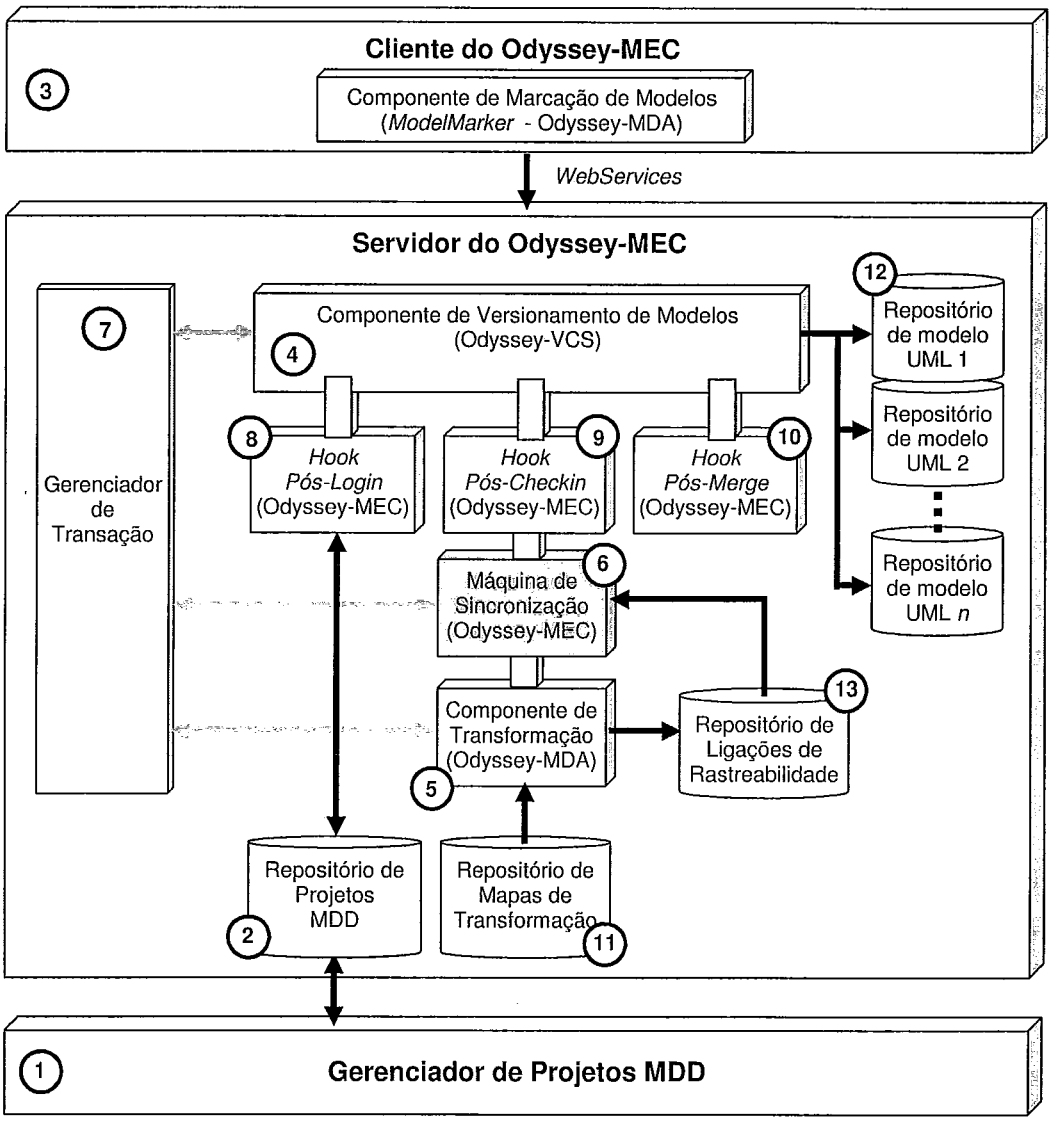

Figura 6.3: Arquitetura do Odyssey-MEC

Além dos componentes descritos, *hooks* são usados para auxiliar a execução do processo. O *hook pós-login* (8), acionado após o usuário ser autenticado, tem como finalidade vincular ao Odyssey-VCS os *hooks pós-checkin* (9) necessários para acionar a máquina de sincronização. Para cada par de modelos do projeto, são usados dois *hooks*, um para a transformação/sincronização esquerda-direita, e outro para a direção oposta. O *hook pós-login* também liga ao Odyssey-VCS o *hook pós-merge* (10), usado para a recuperação de ligações de rastreabilidade após a junção de um elemento.

Os mapas de transformação a serem usados pelo componente de transformação são mantidos em um repositório (11), assim como os modelos, versionados pelo Odyssey-VCS (12), e as ligações de rastreabilidade (13), geradas durante a transformação de modelos. Os detalhes sobre cada parte do Odyssey-MEC são apresentadas nas subseções a seguir.

6.3.1 Repositório de Modelos

Os repositórios de modelos são diretórios do sistema operacional, onde são armazenados os arquivos XMI contendo os modelos e os arquivos XML que mantém as especificações dos mapas de transformação. Uma outra abordagem considerada foi o uso de uma solução baseada em um banco de dados relacional para fazer a persistência dos modelos (ELVER, 2008). Contudo, problemas encontrados na solução inviabilizaram a sua utilização, como, por exemplo, a impossibilidade de gerar a estrutura do banco de dados para a persistência dos modelos UML. Possivelmente os problemas encontrados foram decorrentes da ferramenta ainda não estar em uma versão estável.

Na solução adotada, com o objetivo de manter os modelos organizados, o histórico das versões de cada modelo de um projeto MDD é mantido em um repositório separado. As ligações de rastreabilidade e os mapas de transformação também são mantidos em repositórios distintos.

A manipulação dos modelos existentes nos repositórios é realizada através do *EMFRepository* e seus descendentes (Figura 6.4). Estas classes simplificam a persistência e o acesso aos modelos. A classe *ResourceSet* representa uma coleção de modelos e cada *Resource* representa um modelo específico. A partir do momento que fazem parte do mesmo *ResourceSet*, os modelos podem fazer referência a elementos que estão em outros modelos. A classe *UMLRepository* estende a *EMFRepository* com

o objetivo de facilitar o uso dos perfis UML usados pela abordagem, mantidos por *UMLProfile*.

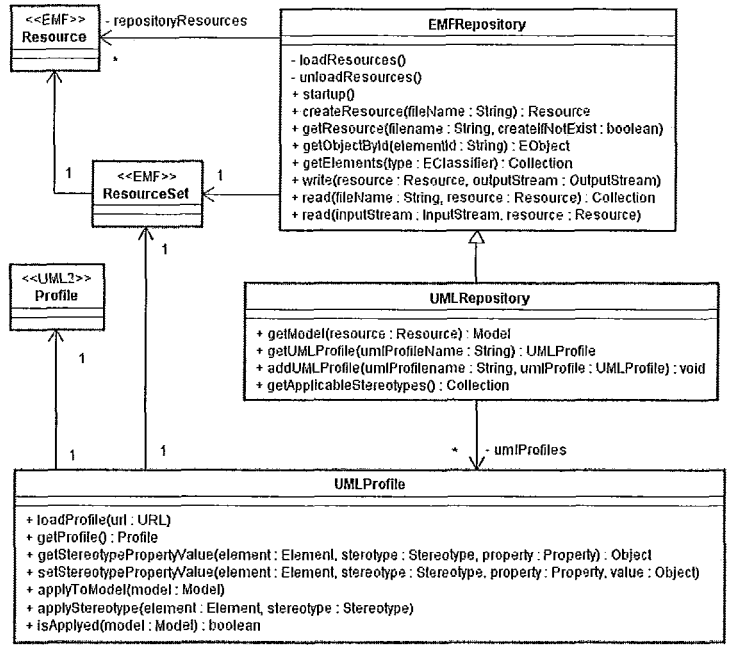

Figura 6.4: Gerenciadores de repositório

6.3.2 Versionamento de Modelos

O controle de versão é usado no Odyssey-MEC para possibilitar o controle da evolução dos modelos de um projeto MDD ao longo do tempo. Além de manter o histórico das modificações de cada modelo, é possível saber as causas, quando ocorreram as alterações e as pessoas responsáveis.

Para a realização do controle de versão, o Odyssey-VCS (OLIVEIRA *et al.*, 2005) foi reutilizado como um componente. Esse componente possui os recursos necessários para controlar a evolução de modelos durante o desenvolvimento e manutenção, como, por exemplo, a junção de versões e operações de *check-in* e *check-out*. Esse componente é um sistema de versionamento com granuralidade fina e tem capacidade de versionar qualquer elemento UML (MURTA *et al.*, 2008). Contudo, devido às particularidades do Odyssey-MEC, foi necessário fazer as seguintes alterações estruturais e comportamentais (MURTA *et al.*, 2008): (1) substituição do MDR (MATULA, 2003) pelo EMF; (2) criação de uma fachada genérica para operações de versionamento; e (3) criação de um mecanismo para execução de *hooks pós-login*, *pós-checkin* e *pós-merge*.

A versão anterior do Odyssey-VCS utilizava o MDR (MATULA, 2003) como a infra-estrutura para manter e gerar versões de modelos. Dado que o MDR manipula modelos UML 1.4 e foi descontinuado, foi necessário migrar o Odyssey-VCS para o EMF, para que fosse possível trabalhar com a UML 2.

O Odyssey-VCS possui um conjunto de operações comuns aos sistemas de controle de versão, como o *check-in* e *check-out* e listagem do conteúdo de um repositório. Essas funcionalidades são representadas em uma interface Java, usada como referência para a implementação das classes que fazem a comunicação cliente/servidor através de *Web Service*. Como o Odyssey-MEC também foi elaborado para operar no mesmo contexto, a interface foi generalizada, de modo a ser usada pelas duas ferramentas. Assim, como pode ser observado na Figura 6.5, a interface *ModelVCS* define os métodos de controle de versão implementados pelas classes *OdysseyVCSWebServices* e *OdysseyMECWebServices*. Assim, o cliente do Odyssey-MEC pode se comunicar com o servidor do Odyssey-MEC e Odyssey-VCS. Esse recurso possibilita, por exemplo, que um modelo existente em um repositório do Odyssey-VCS seja exportado através do cliente do Odyssey-MEC, tornando desnecessário o uso dos dois clientes.

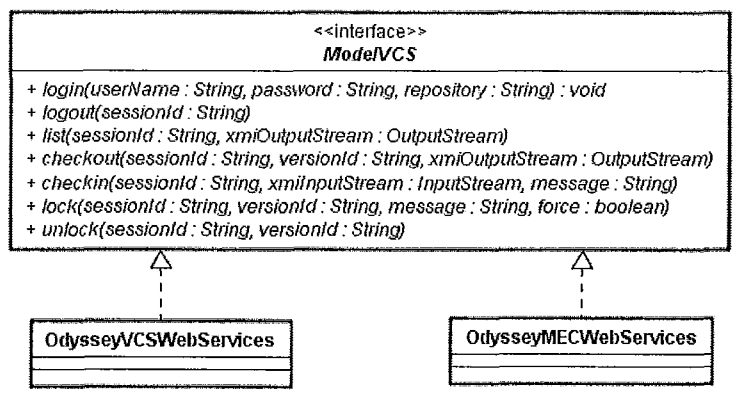

Figura 6.5: Interface genérica de operações de controle de versão

Hooks são recursos usados, por exemplo, pelo Subversion (COLLINS-SUSSMAN *et al.*, 2004) e CVS (CEDERQVIST, 2005), para possibilitar a extensão das operações, como o *check-in* e *check-out*. O Odyssey-MEC precisava de uma forma de acionar automaticamente a máquina de sincronização quando o usuário fizesse o *check-in* de um modelo. Além disso, era necessária uma solução que possibilitasse a propagação da transformação e da sincronização para todos os modelos de um projeto MDD. Para suprir essas necessidades, os seguintes *hooks* foram acrescentados ao

Odyssey-VCS: *pós-login*, *pós-checkin*, e *pós-merge*. O *hook pós-login* é executado sempre que um usuário faz o *login* em um repositório. O *hook pós-checkin* é acionado no fim da operação de *check-in*. Finalmente, o *hook pós-merge* é acionado após a junção de um elemento do modelo.

O mecanismo de *hooks* do Odyssey-VCS foi elaborado de forma que várias tarefas possam ser executadas. Nesse caso, cada tarefa deve ser realizada por uma instância diferente de um *hook*. Esse recurso é essencial para a sincronização de vários modelos-alvo a partir de um mesmo modelo-fonte. Por exemplo, quando o usuário faz o *check-in* de um PIM, um *hook pós-checkin* é usado para sincronizá-lo com um PSM-JEE, enquanto outro pode ser usado para fazer a sincronização com um PSM-NET. A implementação dos *hooks* do Odyssey-MEC será descrita na Seção 6.3.4 .

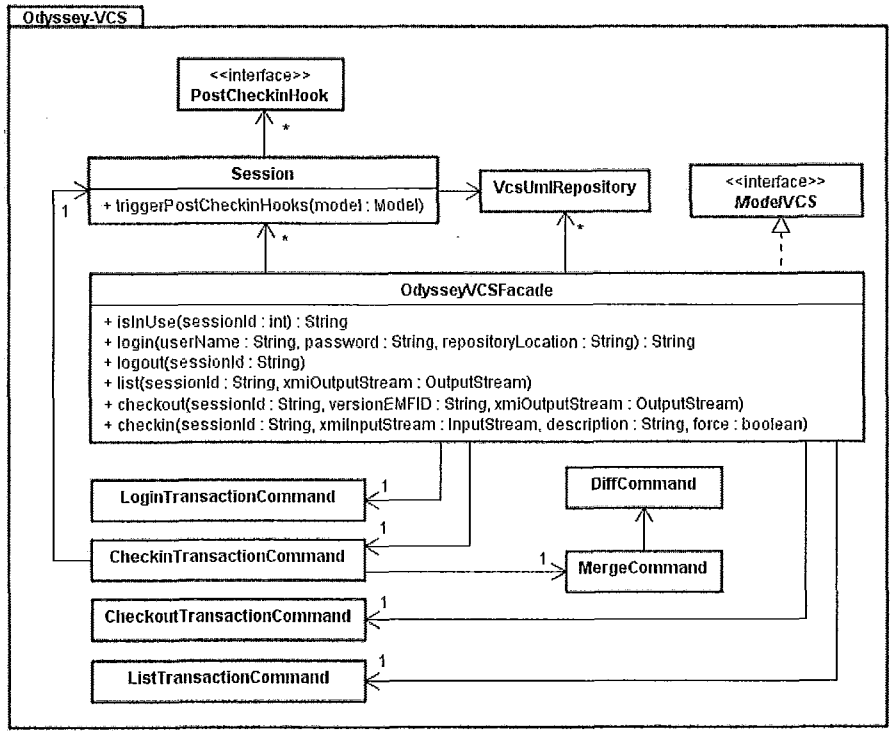

Figura 6.6: Visão parcial na nova implementação do Odyssey-VCS

Uma visão parcial da arquitetura do Odyssey-VCS após as alterações mencionadas é apresentada na Figura 6.6. As classes apresentadas na figura possuem as seguintes finalidades:

- *ModelVCS*: Interface genérica, usada como referência para a implementação de fachadas com os métodos relacionados com controle de versão.

- *OdysseyVCSFacade*: Disponibiliza as operações para a autenticação de usuário (*login*), *check-in*, *check-out*, listagem do conteúdo de um repositórios (*list*) e verificação do uso do repositório (*isInUse*).
- *Session*: Controla uma seção do usuário. Para cada usuário e repositório, é criada uma nova seção.
- *VCSUMLRepository*: Controla a persistência e o acesso de modelos UML versionados. Os dados de versionamento são mantidos junto com o modelo.
- *LoginTransactionCommand*: Executa o login dentro do contexto de uma transação.
- *CheckinTransactionCommand*: Executa o *check-in* dentro do contexto de uma transação.
- *CheckoutTransactionCommand*: Executa o *check-out* dentro do contexto de uma transação.
- *ListTransactionCommand*: Executa a listagem do conteúdo do repositório dentro do contexto de uma transação.
- *MergeCommand*: Faz a junção de modelos.
- *DiffCommand*: Identifica as diferenças entre os modelos.

6.3.3 Transformação de Modelos

A transformação permite ao Odyssey-MEC a criação automática de modelos. O componente usado para essa tarefa é o Odyssey-MDA (MAIA, 2006). Essa ferramenta permite a transformação de modelos em duas direções, usando como referência um conjunto de mapeamentos. Contudo, assim como o Odyssey-VCS, algumas modificações foram necessárias: (1) substituição do MDR pelo EMF; (2) geração de ligações de rastreabilidade; (3) transformação de tipos simples; e (4) uso de perfis para a aplicação de estereótipos e valores etiquetados.

A estratégia adotada para a sincronização do Odyssey-MEC depende da existência de **ligações de rastreabilidade**. O Odyssey-MDA foi modificado de modo a gerar as ligações. Desse modo, sempre que um elemento é gerado, é criada uma ligação de rastreabilidade relacionando o elemento do modelo-fonte com o elemento criado. As ligações de rastreabilidade são criadas com o auxílio do *TraceHandler* (Figura 6.7). O Odyssey-MDA utiliza mecanismos para fazer a transformação de modelos. Para que a transformação de tipos simples fosse possível, foi criado o mecanismo

`DataTypeDataType`. Essa classe é instanciada pela máquina de transformação com base no mapa de transformação (Figura 6.7).

A versão original do Odyssey-MDA permitia a criação de estereótipos e etiquetas independente de um perfil. Contudo, a partir da UML 2.0, o uso de perfis para representar estereótipos e valores etiquetados tornou-se obrigatório. Assim, foram criados perfis para a representação dos estereótipos e etiquetas usadas como marcações pelo Odyssey-MDA. Desse modo, a aplicação de novas marcações depende da criação de novos perfis.

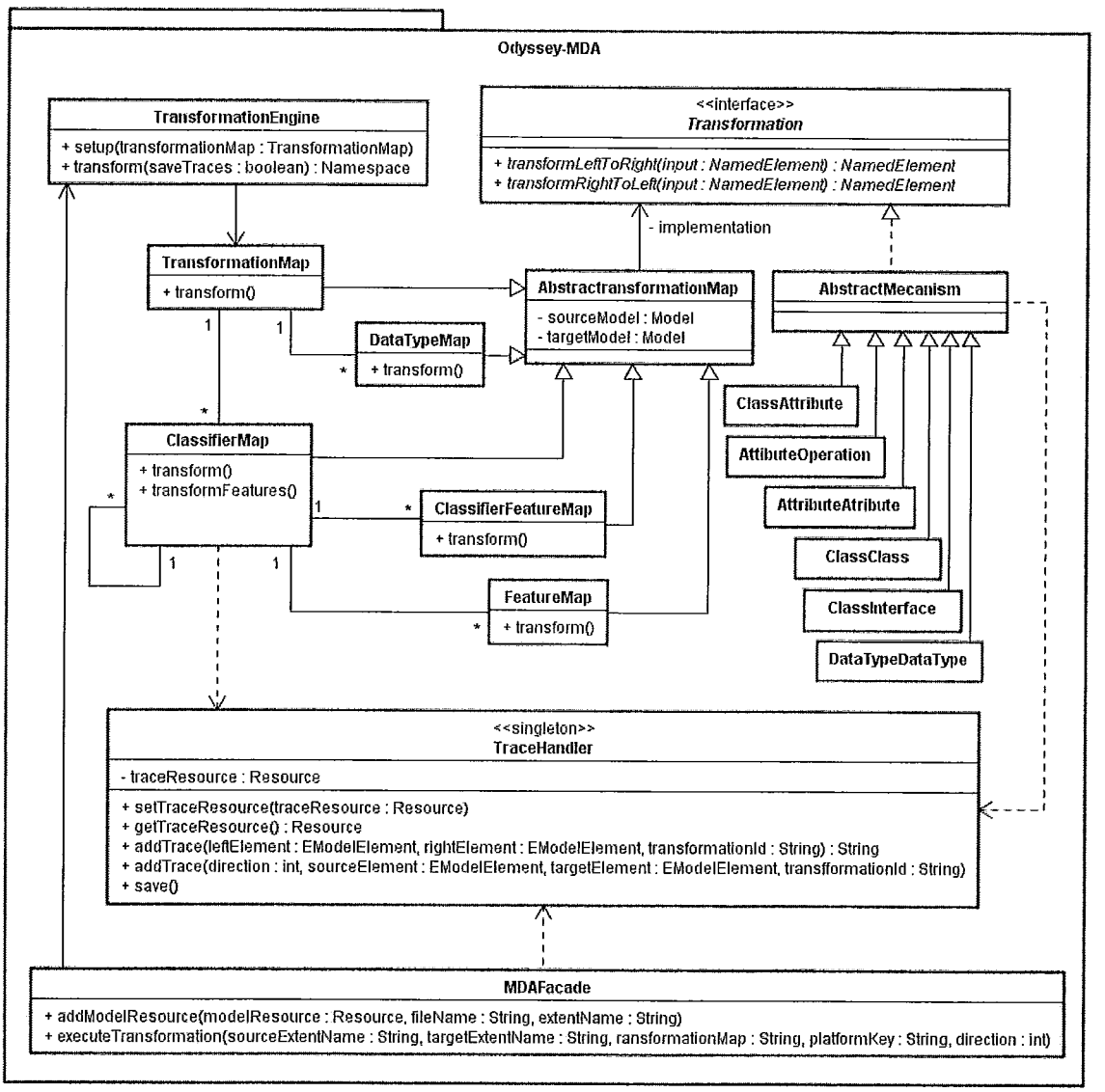

Figura 6.7: Estrutura básica do Odyssey-MDA

A estrutura básica do Odyssey-MDA após as alterações é apresentada na Figura 6.7. Os principais elementos apresentados na figura possuem as seguintes finalidades:

- *TransformationEngine*: Coordena o processo de transformação de modelos.

- *TransformationMap*: Mantém na memória o mapa de transformação a ser usado durante o processo de transformação.
- *DataTypeMap*: Representa um mapeamento de tipo para tipo.
- *ClassifierMap*: Representa um mapeamento entre classificadores, como classes e interfaces, por exemplo.
- *ClassifierFeatureMap*: Representa um mapeamento de um classificador para um atributo ou método (*features*).
- *FeatureMap*: Representa um mapeamento de um atributo ou método para outro atributo ou método.
- *Transformation*: Esta interface representa uma transformação.
- *ClassAttribute*: Faz a transformação de uma classe para atributo, e vice-versa.
- *AttributeOperation*: Faz a transformação de um atributo para um método, e vice-versa.
- *ClassClass*: Faz a transformação de uma classe para outra.
- *ClassInterface*: Faz a transformação de uma classe para interface, e vice versa.
- *DataTypeDataType*: Faz a transformação de um tipo simples para outro.
- *TraceHandler*: Fornece o suporte necessário para a criação de ligações de rastreabilidade.
- *MDAFacade*: Fornece acesso aos métodos que devem ser executados para realizar a transformação de modelos.

6.3.4 Sincronização de Modelos

A máquina de sincronização tem como objetivo fazer a transformação, sincronização e o versionamento dos modelos-alvo. As classes usadas para a realização dessas tarefas estão representadas na Figura 6.8. Essas classes possuem as seguintes finalidades:

- *SynchronizationEngine*: Inicia o processo de sincronização dos modelos.
- *SynchronizationTransactionCommand*: Realiza o processo de transformação, sincronização e versionamento do modelo-alvo, dentro do contexto de uma transação.
- *TraceFinder*: Localiza rastros entre elementos de modelos diferentes.
- *SynchronizationArgument*: Mantém as informações relacionadas à sincronização que está sendo realizada.

- *OdysseyVCSFacade*: Usado para fazer a recuperação da versão anterior do modelo-alvo, se existir, e para fazer o versionamento do modelo-alvo.
- *Session*: Usado para controlar a manipulação de um repositório de modelos. Cada usuário autenticado possui uma seção. E cada seção possui a lista de *hooks pós-checkin* que devem ser executados. No caso do Odyssey-MEC, *Session* mantém uma lista de *OdysseyMECPostCheckinHook.*
- *OdysseyMECPostCheckinHook*: É vinculado a uma seção do Odyssey-VCS, Tem como responsabilidade acionar a máquina de sincronização. (*SynchronizationEngine*).
- *MDAFacade*: Usada para a transformação de modelos.

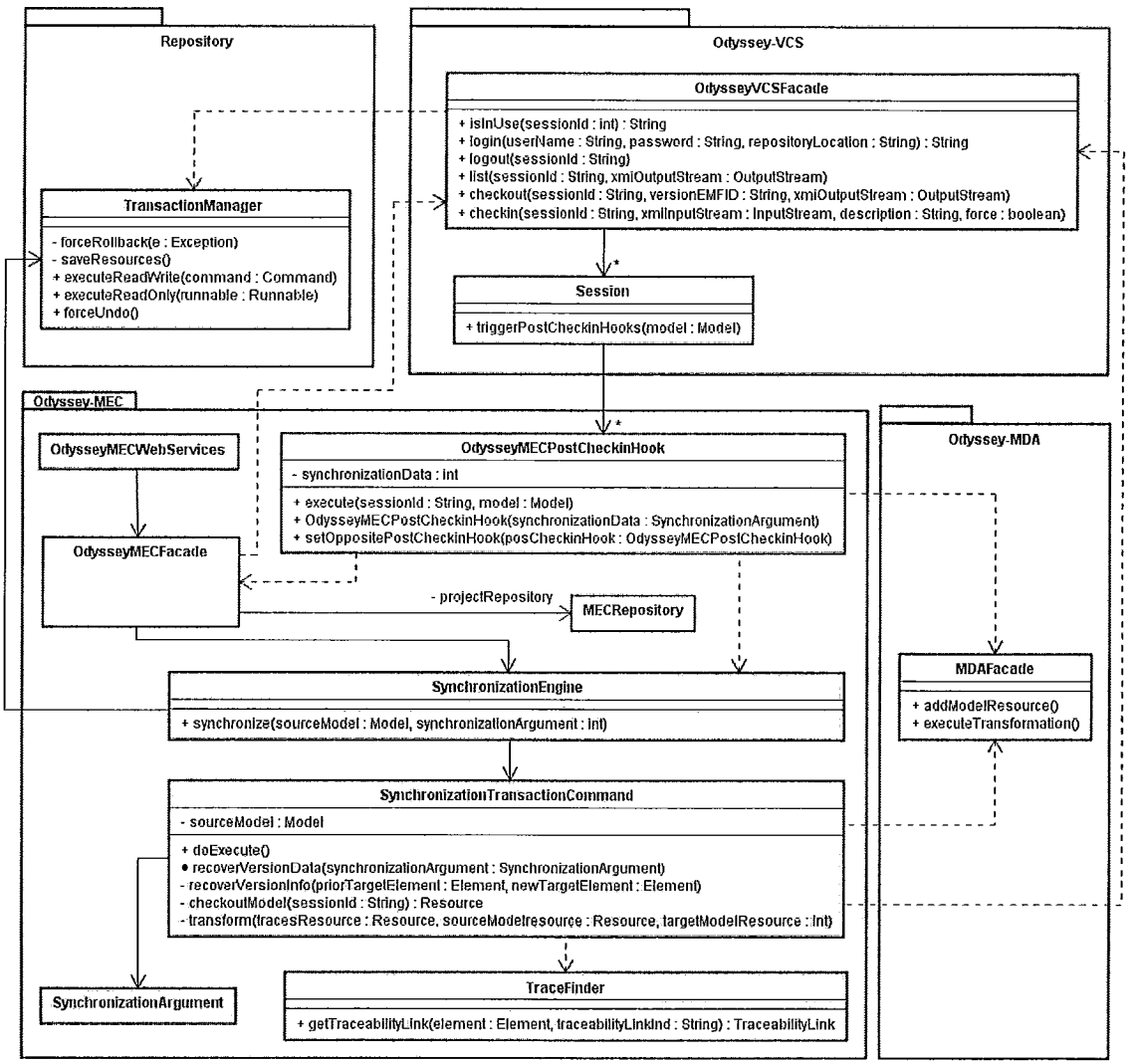

Figura 6.8: Máquina de sincronização e suas dependências

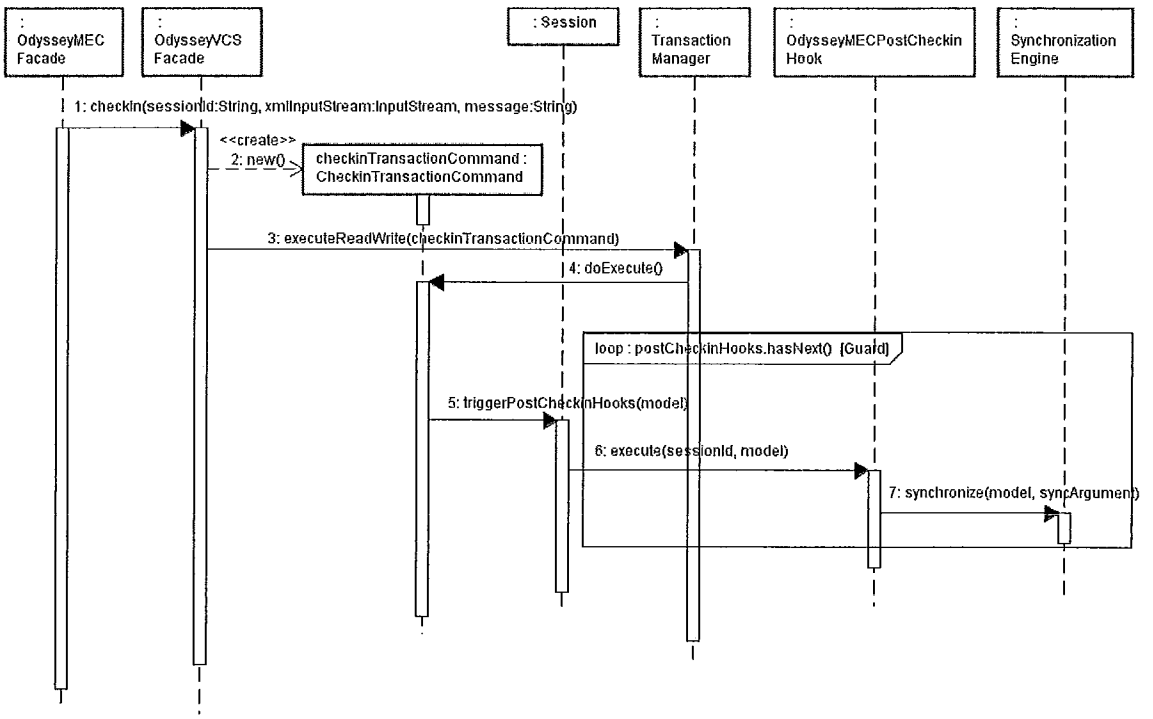

Figura 6.9: Execução da máquina de sincronização a partir do *check-in*

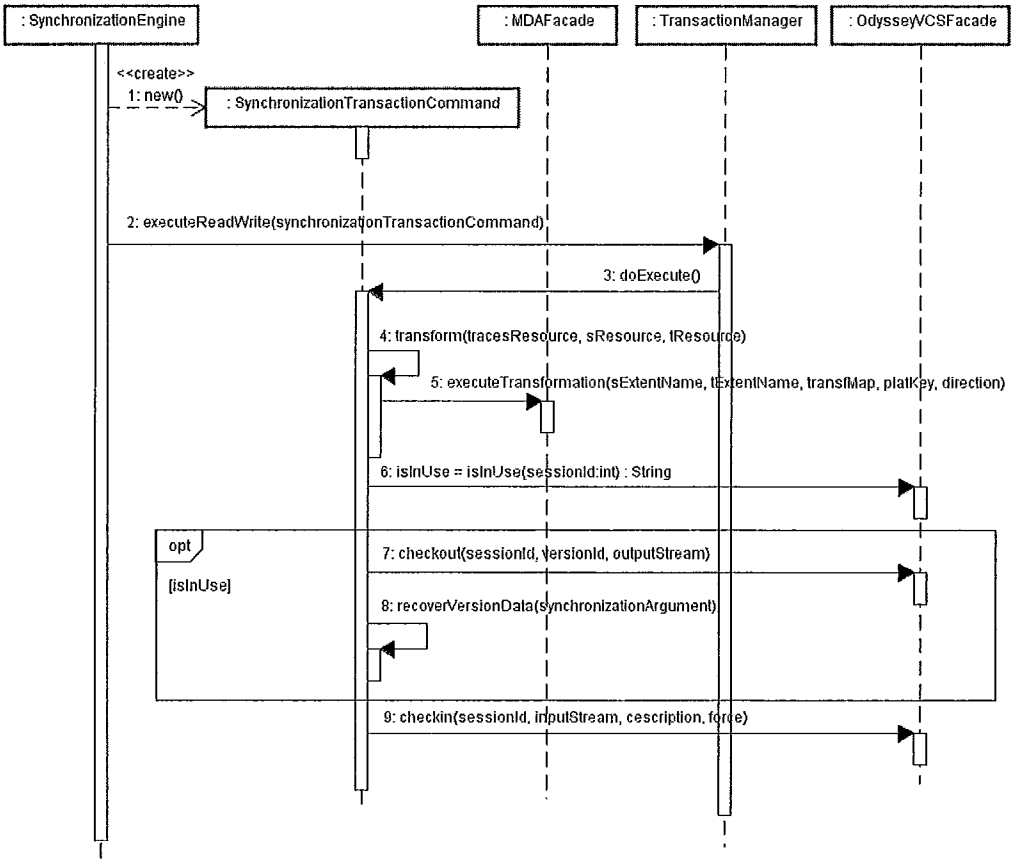

Figura 6.10: Processo de transformação e sincronização

O processamento para a transformação, sincronização e versionamento de modelos é apresentado nos diagramas de seqüência das Figuras 6.9 e 6.10. Quando o engenheiro de software faz o *check-in* do modelo-fonte, o *OdysseyMECWebService* aciona o método *checkin* da fachada *OdysseyMECFacade*. Essa chamada cria uma instância de *CheckinTransactionCommand* para fazer o versionamento do modelo-fonte. A instância é passada para o *TransactionManager*, que tem a finalidade de gerenciar a transação que está sendo realizada. Após o versionamento do modelo, o *CheckinTransactionCommand* envia a mensagem *triggerPostCheckinHooks* para a instância de *Session*. Como resultado, é realizada uma iteração, em que o método *execute* de *OdysseyMECPostCheckinHook* é executado. Esse método envia a mensagem *synchronize para SynchronizationEngine* para a realização do processo de sincronização propriamente dito.

Para realizar a sincronização, *SynchronizationEngine* aloca uma instância de *SynchronizationTransactionCommand* (STC) e a transfere para o *TransactionManager* para execução. Ao ser acionada, STC executa o método *transform*. Esse método faz a chamada de *executeTransformation* de *MDAFacade* para fazer a transformação propriamente dita. Em seguida, STC chama o método *isInUse* de *OdysseyMDAFacade* para verificar se o repositório já possui uma versão do modelo-alvo gerado a partir da transformação. Se existir, STC faz o *check-out* do modelo e, em seguida, executa o método *recoverVersionData* para recuperar as informações de versionamento do modelo. No final do processamento, STC executa o método *checkin* de *OdysseyVCSFacade* para fazer o *check-in* do modelo-alvo. Se o modelo-alvo estiver relacionado a outro modelo, o processo descrito se repete.

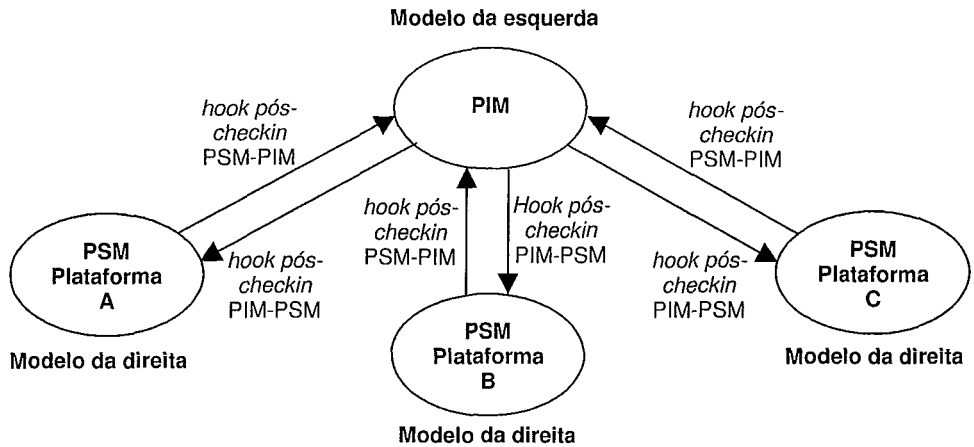

Figura 6.11: Hooks para transformação esquerda-direita e direita-esquerda

O processo de transformação e sincronização de modelos pode ser realizado em duas direções. Por esse motivo, para cada par de modelos (modelo da esquerda e modelo da direita), são usados dois *hooks pós-checkin*: um para transformar e sincronizar o modelo da direita com o modelo da esquerda (transformação/sincronização *esquerda-direita*) e outro para realizar o processo inverso (transformação/sincronização *direita-esquerda*). A Figura 6.11 apresenta um exemplo de configuração dos *hooks* para sincronização dos modelos de um projeto MDD.

Considerando que o mecanismo de sincronização do *Odyssey-MEC* faz chamadas ao *Odyssey-VCS*, a existência dos pares de *hooks* opostos poderia provocar uma execução recursiva infinita dos *hooks*. Por exemplo, quando o *hook* PIM-PSM para a *plataforma A* fosse executado, o mecanismo de transformação/sincronização do *Odyssey-MEC* seria acionado. Após a criação do novo PSM para a *plataforma A* e a recuperação dos dados de versionamento, o mecanismo de transformação/sincronização chamaria o *Odyssey-VCS* para fazer o *check-in* do PSM, o que acionaria o *hook* PSM-PIM da *plataforma A* para gerar o PIM. Após a geração desse modelo, o *Odyssey-MEC* acionaria novamente o *Odyssey-VCS* para fazer o versionamento do PIM, o que iria disparar novamente o *hook* PIM-PSM da *plataforma A*. Esse processo seria mantido infinitamente. Para evitar esse problema, sempre que um *hook* é acionado, o *hook* da transformação oposta é desativado, permanecendo neste estado até o término da operação.

6.3.5 Controle de Transação

O Odyssey-MEC foi elaborado com a finalidade de manter modelos diferentes consistentes. Para isso, são realizadas operações de versionamento, transformação e recuperação de dados de versionamento. Durante o processamento, são geradas novas versões de modelos e novas ligações de rastreabilidade. Considerando a possibilidade de falha em um dos sub-processos, foi criado um gerenciador de transação (*TransactionManager*) (Figura 6.12) para garantir a consistência dos repositórios. Esse componente é baseado no *EMF Transaction* (ECLIPSE, 2008c).

O *EMF Transaction* permite a manipulação de modelos dentro do contexto de transações aninhadas. Desse modo, uma transação pode ser realizada dentro de outra, possibilitando a realização de transações em várias fases. Se a transação mais interna falhar, as transações externas são canceladas.

Para manipular modelos dentro de uma transação, é necessário criar um comando descendente de *RecordingCommand*. Assim, para realizar transações aninhadas, é necessário um comando para cada transação. Esses comandos devem ser passados para o *TransactionCommandStack* do *EMF Transaction*.

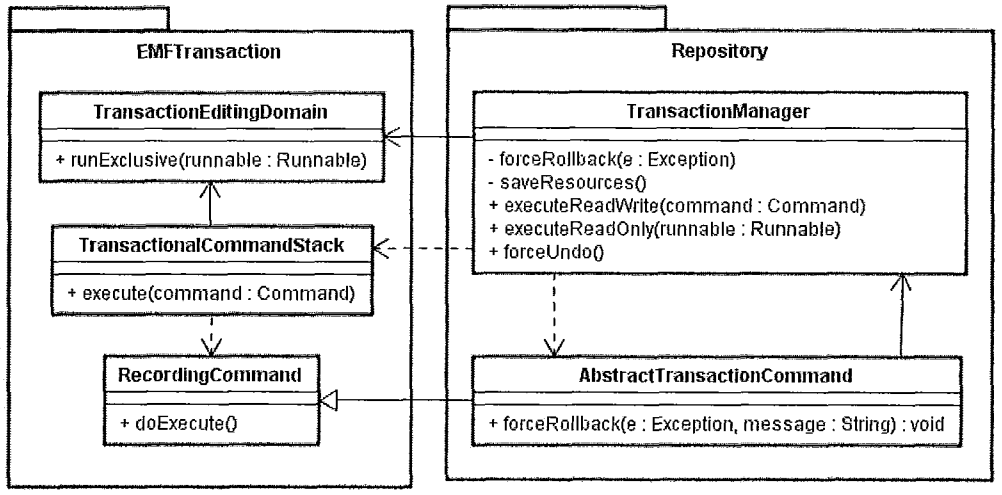

Figura 6.12: Gerenciador de transações

O gerenciador de transações atua como um intermediário, simplificando a execução de comandos com transação e aumentando o controle sobre as transações. Por exemplo, as alterações são gravadas somente quando todas as transações são encerradas com sucesso. A classe *AbstractTransactionCommand* simplifica a propagação de cancelamento de transações realizadas a partir de exceções.

6.3.6 Comunicação Cliente-Servidor

O Odyssey-MEC foi projetado com uma arquitetura cliente-servidor. A comunicação é realizada através de *Web Services* (BOOTH *et al.*, 2005). A implementação *Web Services* usada é o *Axis* (APACHE, 2008b), que é executado sobre o *Tomcat* (APACHE, 2008a), um servidor de aplicações para *Web* (Figura 6.13). Quando o cliente envia uma requisição de serviço para o *Tomcat*, a mensagem é repassada para o *Axis*, que se encarrega de identificar a classe que implementa o serviço e de executá-lo. No caso do Odyssey-MEC, essa classe é a *OdysseyMECWebServices* (Figura 6.14).

Quando o engenheiro de software realiza uma operação de *check-in*, o cliente envia para o servidor o arquivo XMI contendo o modelo a ser versionado junto com a requisição. Quando a operação realizada é o *check-out*, o servidor envia um arquivo XMI contendo o modelo para o cliente.

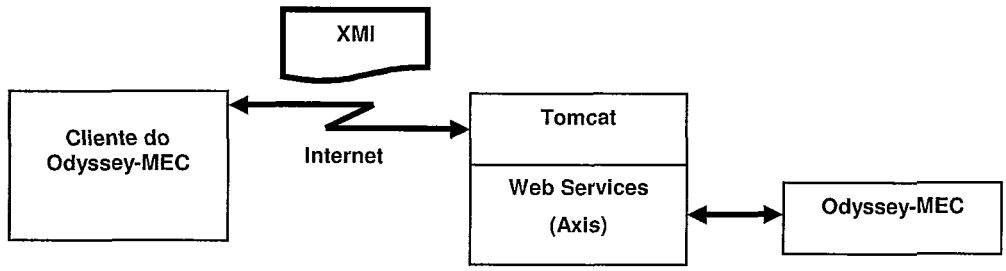

Figura 6.13: Comunicação cliente-servidor do Odyssey-MEC

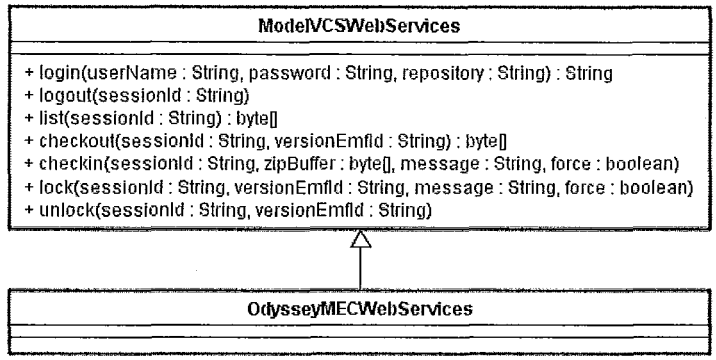

Figura 6.14: *Web Services* do Odyssey-MEC

6.4 Uso do Protótipo

Na seção anterior, foram descritos os principais aspectos da implementação do protótipo do Odyssey-MEC. Nesta seção, é apresentado o uso do protótipo a partir do gerenciador de projetos e do cliente do Odyssey-MEC. Exemplos relacionados com o controle de evolução de modelos a partir da abordagem são apresentados na Seção 6.5.

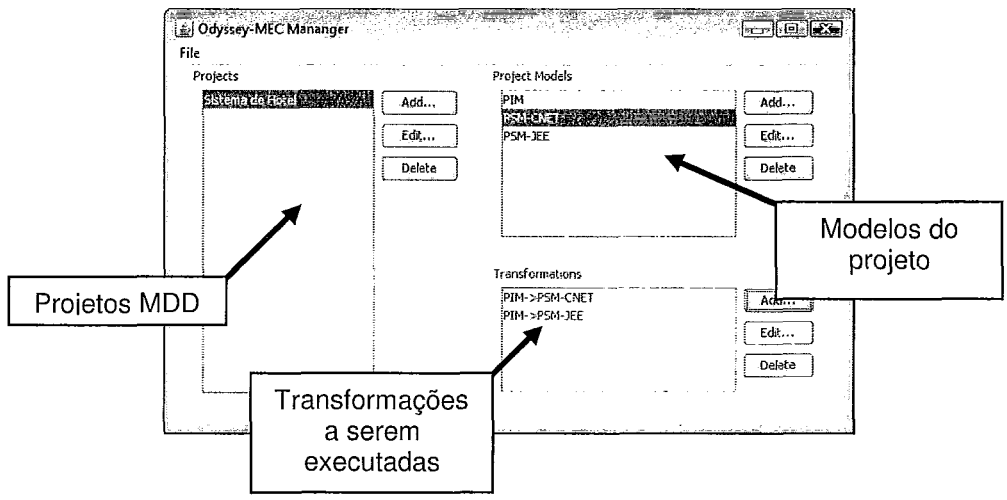

Figura 6.15: Tela principal do Odyssey-MEC Manager

6.4.1 Preparação do Projeto

A primeira etapa para o controle da evolução dos modelos é a criação de um projeto MDD. Para isso, é usado o *Odyssey-MEC Manager* (Figura 6.15). A partir dessa ferramenta, são definidos: (1) o nome e a localização do projeto (Figura 6.16); (2) os modelos, suas respectivas plataformas, quando for o caso, e diretórios (Figura 6.17); e (3) as transformações a serem executadas (Figura 6.18). Vale a pena notar que a definição de transformações é caracterizada pela seleção dos modelos da esquerda e da direita, e do mapa de transformação a ser usado. Além disso, as plataformas e os mapas de transformação são cadastrados a partir do próprio gerenciador. Os cadastros de plataformas e de mapas de transformação são disponibilizados a partir dos comandos *Plataforms* e *Transformation Maps*, respectivamente, presentes no menu *File*.

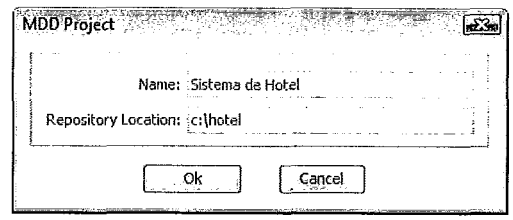

Figura 6.16: Tela de especificação do projeto

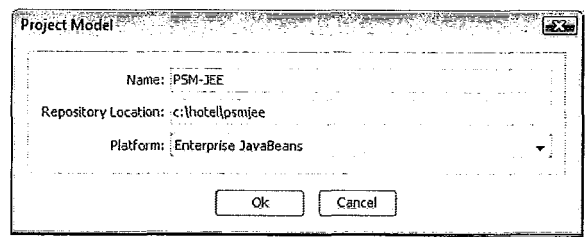

Figura 6.17: Tela de especificação de modelo

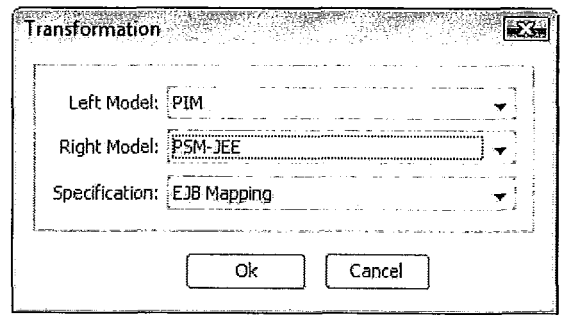

Figura 6.18: Tela de especificação de transformação

6.4.2 Realizando o Login

Após a configuração do projeto, o aplicativo cliente do Odyssey-MEC pode ser usado para o controle da evolução dos modelos. Para realizar as tarefas disponibilizadas pelo Odyssey-MEC, o engenheiro de software deve fazer o *login* no repositório do respectivo modelo. Após o *login*, o Odyssey-MEC prepara o Odyssey-VCS para acionar os *hooks* que irão possibilitar e execução da transformação e sincronização dos modelos do projeto (Figura 6.19).

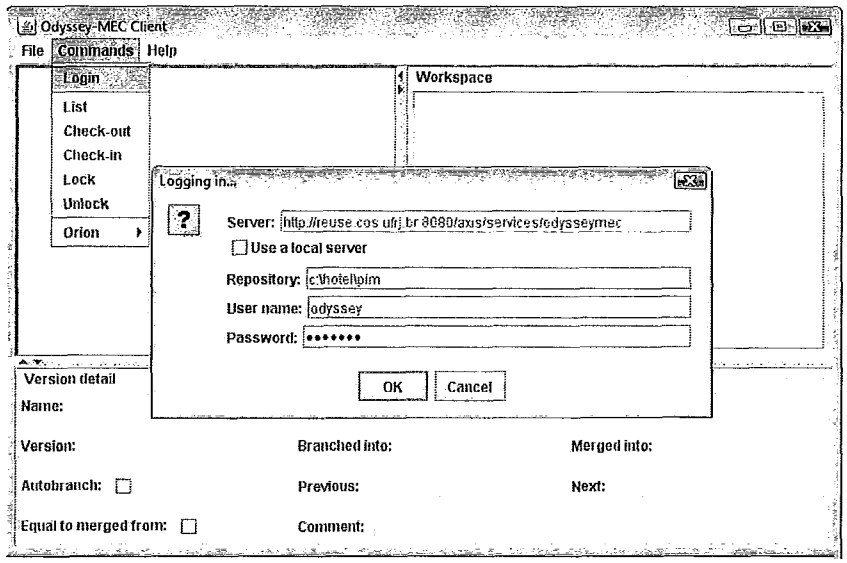

Figura 6.19: *Login* em um repositório

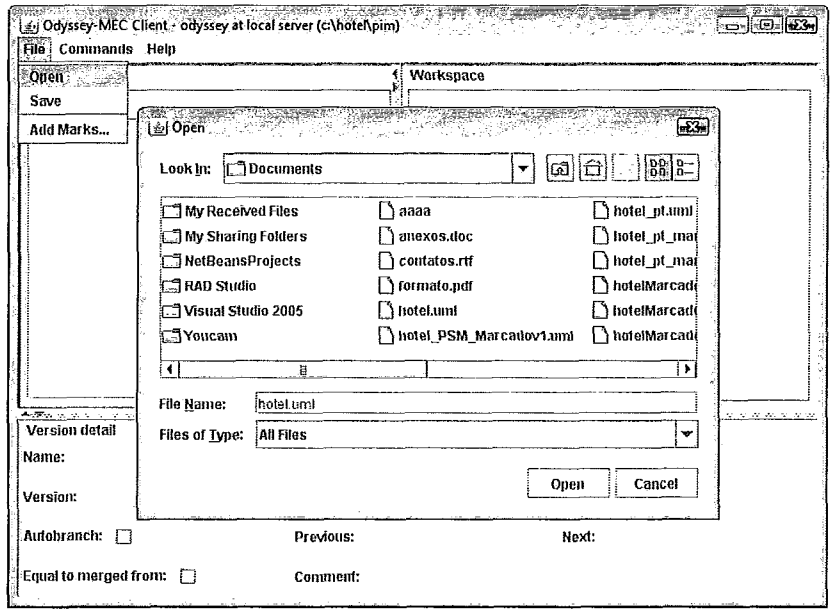

Figura 6.20: Importação do modelo

6.4.3 Importação de um Modelo para o Cliente do Odyssey-MEC

Depois que o engenheiro de software está autenticado, a próxima tarefa é abrir o modelo no espaço de trabalho do aplicativo cliente (Figura 6.20). Vale a pena ressaltar que o modelo deve estar representado com o formato XMI 2.1.

Depois que o modelo é aberto, é possível aplicar as marcações, se necessário, e transferir do modelo para o repositório através da operação de *check-in*, como será apresentado mais adiante.

6.4.4 Marcação de Modelos

Para que o Odyssey-MEC possa fazer a transformação de modelos, é necessário marcar os elementos de acordo com os mecanismos de transformação do Odyssey-MDA. Para facilitar esta tarefa, o Odyssey-MEC utiliza o *Model Marker* (Figura 6.21) (MAIA, 2006) como componente. A partir do *Model Marker*, o engenheiro de software pode selecionar vários elementos e aplicar as marcações necessárias em todos de apenas uma vez. No exemplo da Figura 6.21, o *Model Marker* está sendo usado para atribuir o estereótipo `<<Entity>>` para as classes de entidade (*TipoQuarto*, *Convidado*, *Reserva*, *Cliente*, *Hotel* e *Quarto*).

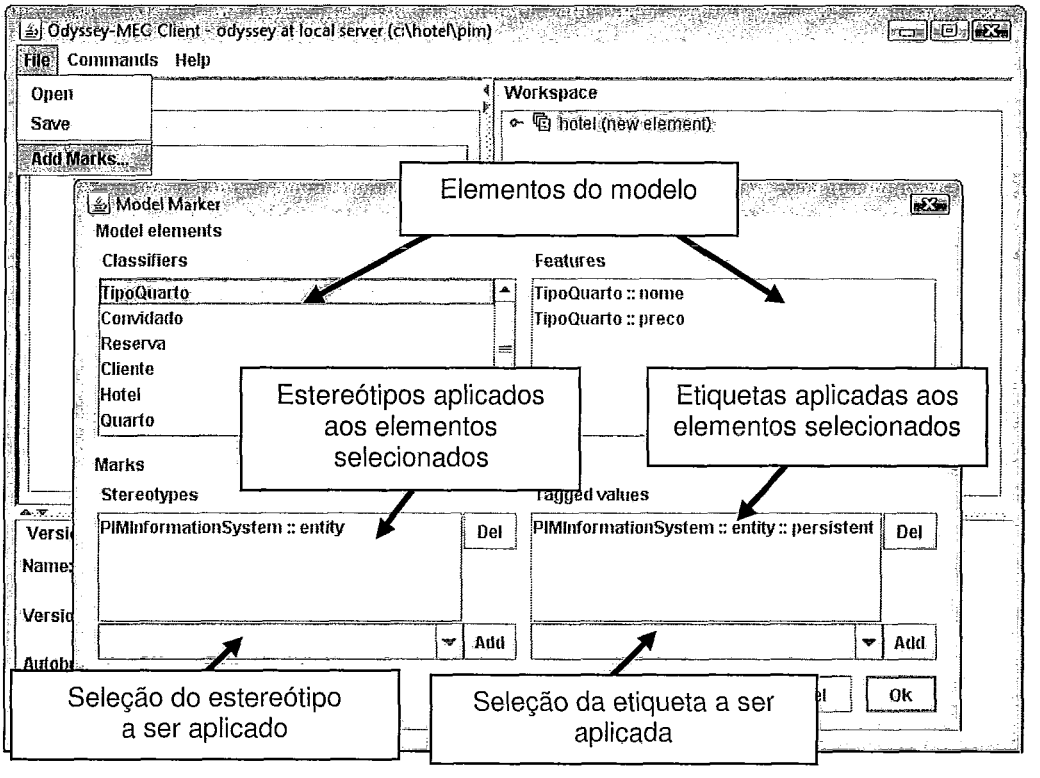

Figura 6.21: Model Marker (MAIA, 2006)

6.4.5 Check-in de um Modelo

Para fazer o *check-in* de um modelo, o usuário deve executar o comando *Operations->Checkin*. Essa operação abre a tela onde é possível descrever as alterações realizadas e forçar a criação de uma nova versão do modelo, caso existam conflitos (Figura 6.22). Após *o check-in*, a transformação é realizada e o modelo-fonte é exibido no painel *Repository* (Figura 6.23).

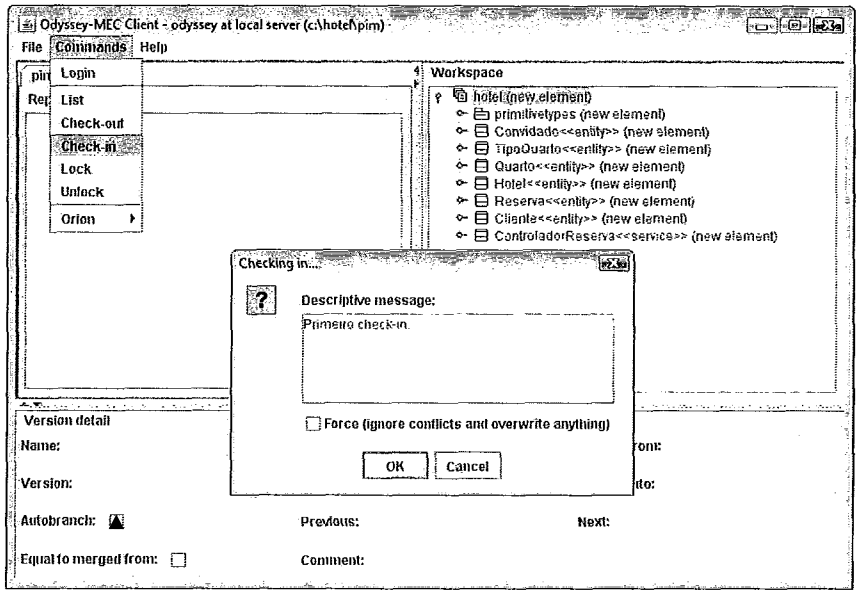

Figura 6.22: Check-in de um modelo (PIM)

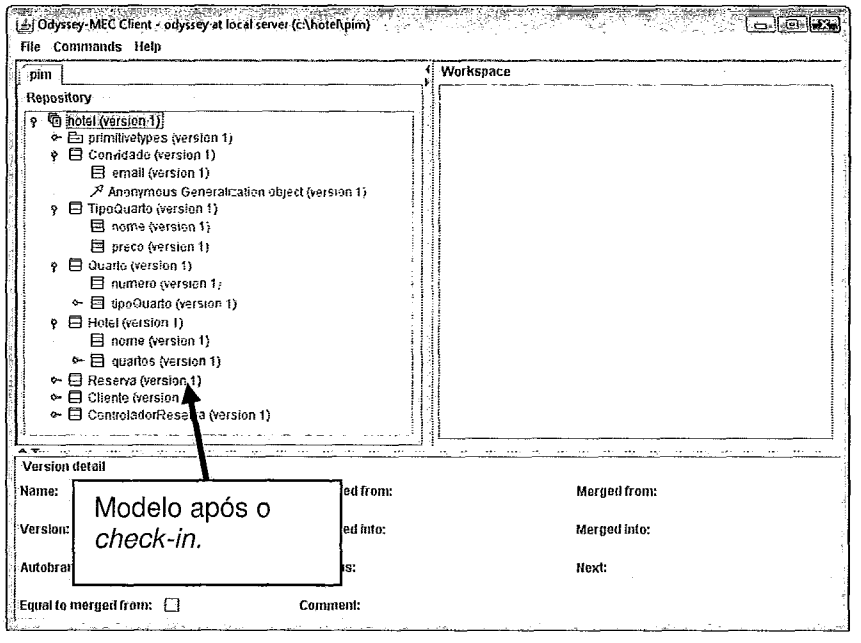

Figura 6.23: Modelo-fonte versionado após o check-in

6.4.6 Check-out de um Modelo

Para fazer o *check-out* de um modelo, o usuário deve selecionar o modelo exibido na caixa *Repository* e executar o comando *Operations->Checkout.* Essa operação disponibiliza o modelo da caixa Workspace do cliente do Odyssey-MEC (Figura 6.24).

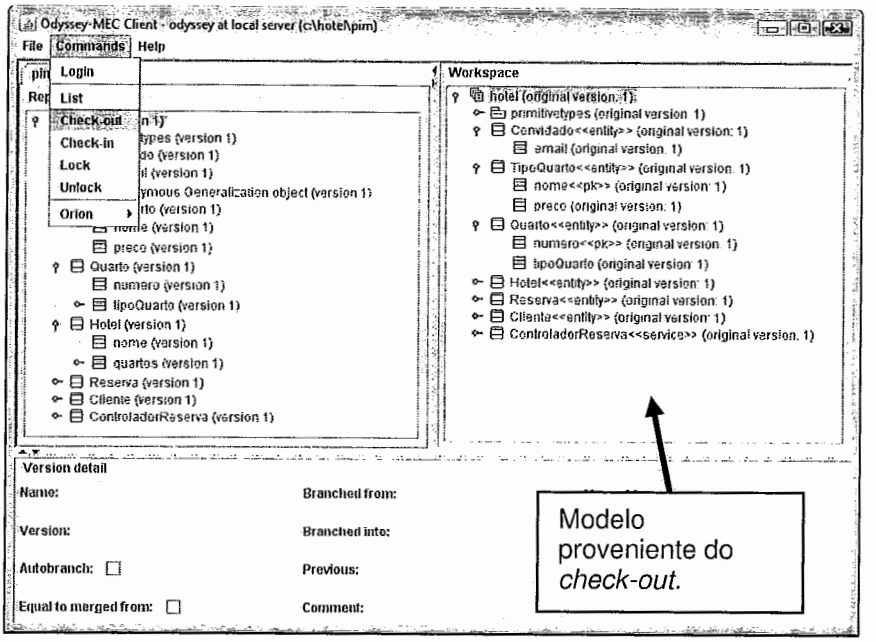

Figura 6.24: Versão obtida a partir da operação de check-out

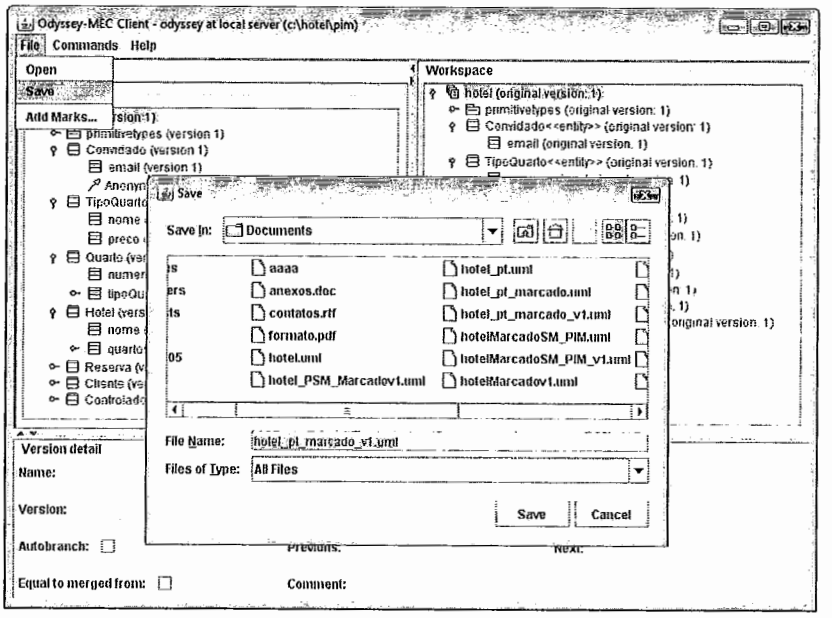

Figura 6.25: Exportação de um modelo

6.4.7 Exportação de um Modelo

Finalmente, para manipular o modelo em uma ferramenta CASE, é necessário exportar o modelo exibido no espaço de trabalho do cliente do Odyssey-MEC. Para isso, é necessário executar o comando *File->Save* e salvar o arquivo em um diretório.

6.4.8 Visualização de Vários Modelos

Apesar de normalmente um desenvolvedor acessar apenas o seu modelo, é possível visualizar vários modelos ao mesmo tempo. Para isso, é necessário apenas acessar o repositório do modelo desejado. Nesse caso, cada modelo é exibido em um painel de repositório diferente, como pode ser observado na Figura 6.26.

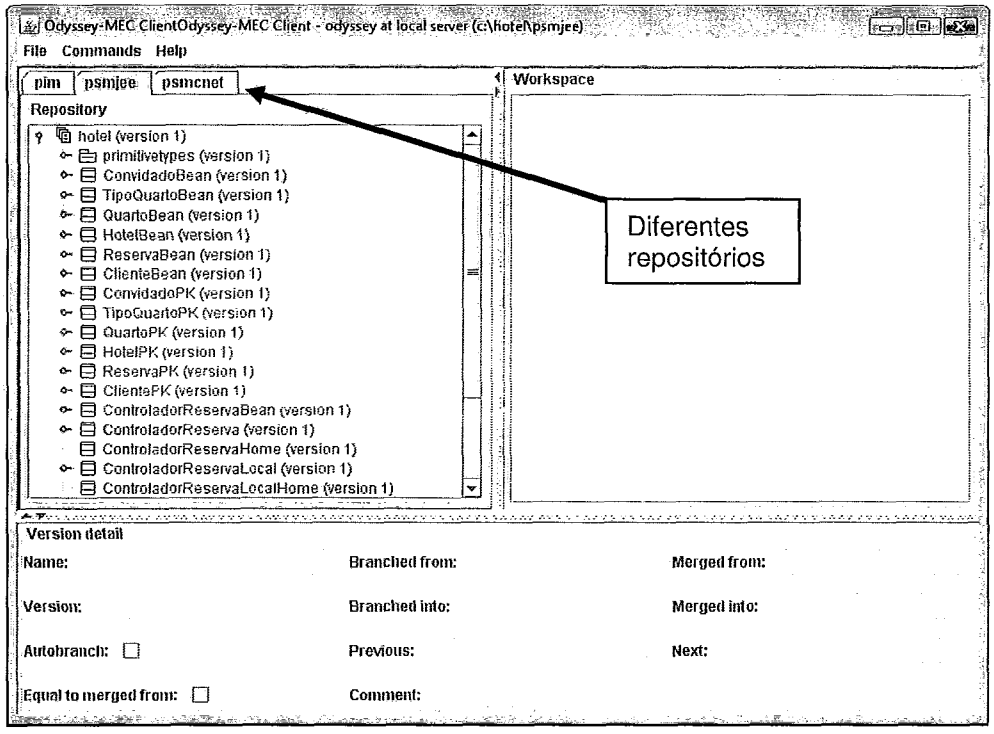

Figura 6.26: Visualização de mais de um repositório

6.5 Exemplos de Uso do Protótipo

Para demonstrar o uso do protótipo do Odyssey-MEC, esta seção apresenta alguns exemplos baseados nos modelos da Figura 6.27. Assim como descrito no Capítulo 5, o modelo de casos de uso e o modelo de classes representam um sistema para controle de reservas de um hotel. Nos exemplos, serão criados *PSMs* do sistema para as plataformas *JEE* e *C#* com *.NET*.

Como cenário, está sendo considerada a participação de três desenvolvedores geograficamente distribuídos: Maria, localizada no Rio de Janeiro e responsável pelo PIM; João, localizado em São Paulo e responsável pelo PSM-JEE; e Pedro, localizado em Recife e responsável pelo PSM-CNET. O projeto já está devidamente configurado, de modo que o cliente do Odyssey-MEC pode ser usado para a realização das operações de *check-in* e *check-out*.

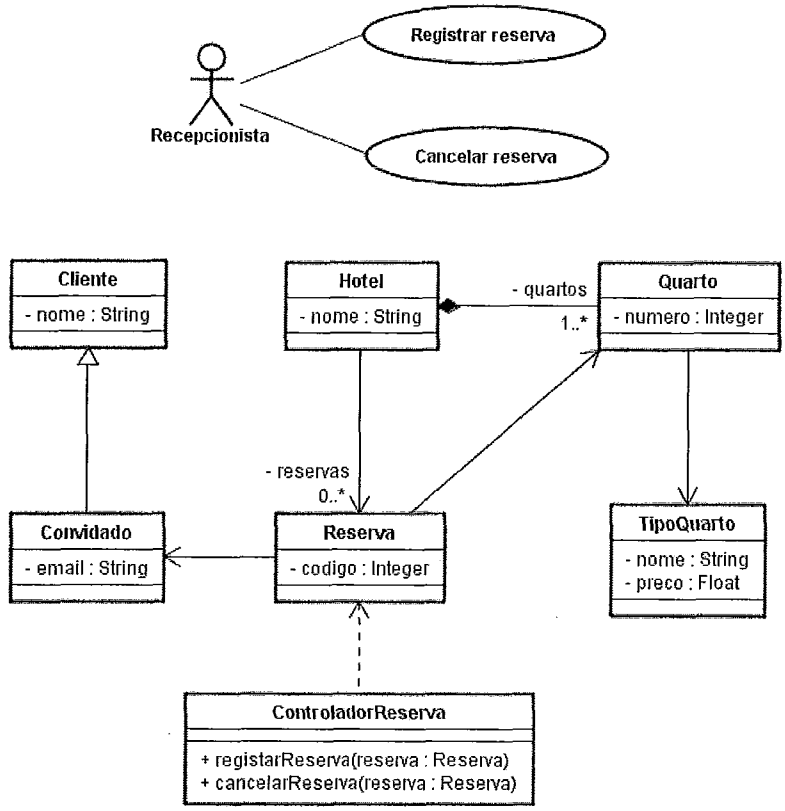

Figura 6.27: Modelo de casos de uso de classes (PIM) usado nos exemplos

6.5.1 Exemplo 1: Primeiro Check-in do PIM

Ao terminar a criação do PIM, Maria utiliza o cliente do Odyssey-MEC para executar o *check-in* do modelo, que é transmitido para o servidor, onde ocorre o versionamento e a transformação do PIM, gerando os modelos PSM-JEE e PSM-CNET (Figura 6.28). Vale a pena notar que, neste exemplo, foram realizadas duas transformações esquerda-direita. Como é o primeiro *check-in*, a operação de sincronização não é realizada. A partir desse momento, os PSMs estão automaticamente disponíveis para João e Pedro.

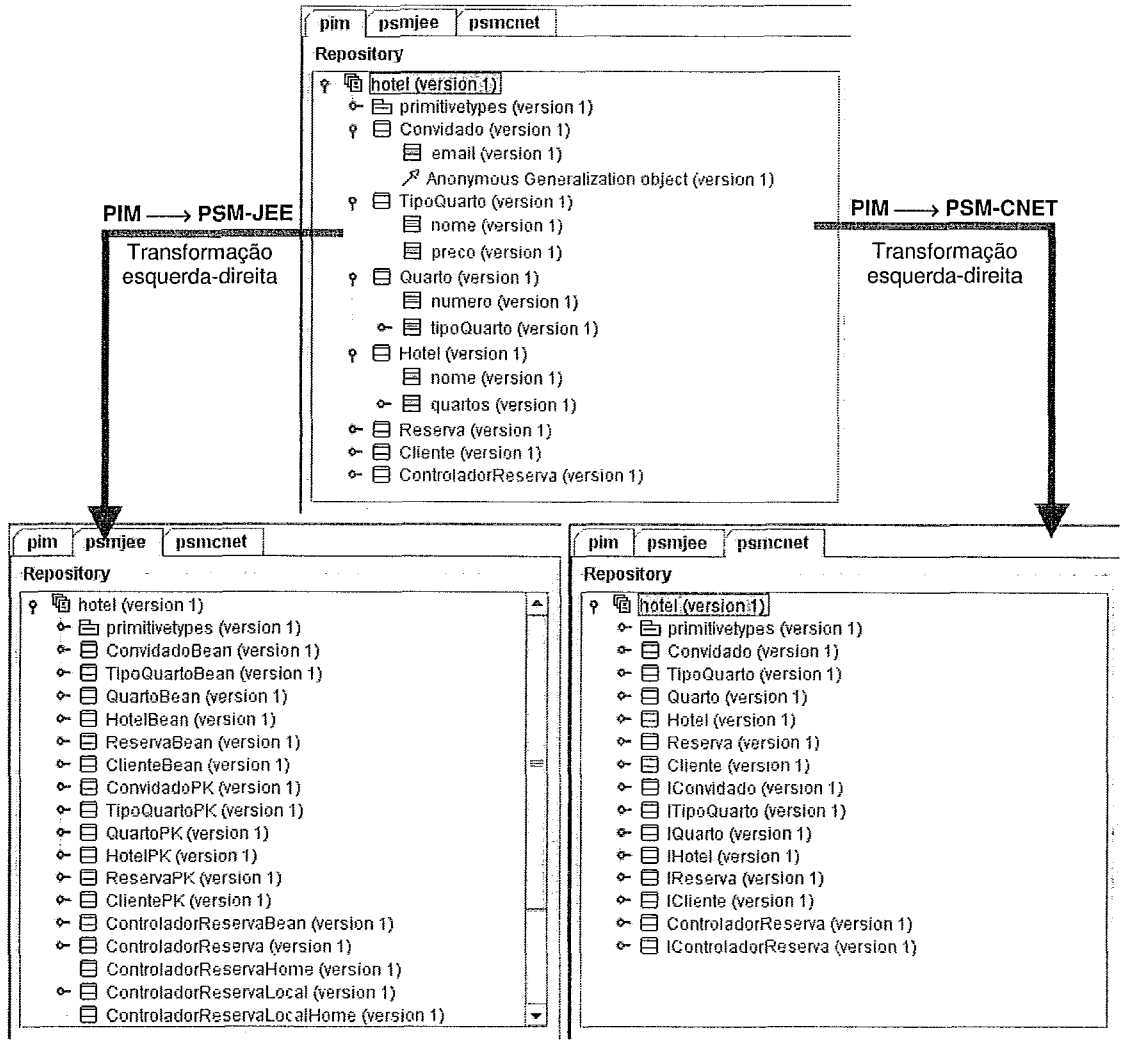

Figura 6.28: PIM e PSMs após o primeiro *check-in* do PIM

6.5.2 Exemplo 2: Modificação do PIM

Durante o desenvolvimento, Maria percebe que precisa fazer as seguintes alterações no PIM: inclusão do atributo *telefone* na classe *Cliente* e alteração do nome do atributo *preco* da classe *TipoQuarto* para *precoIndividual*. Para isso, Maria faz o *check-out*, exporta o modelo e faz as alterações. Em seguida, a engenheira faz o *check-in* do modelo. Como resultado, os modelos PSM-JEE e PSM-CNET são sincronizados com o PIM, conforme pode ser observado na Figura 6.29. Vale a pena notar as novas versões dos elementos dos PSMs correspondentes aos elementos do PIM que foram modificados (*preco*, *getPreco*, *setPreco* para *precoIndividual*, *getPrecoIndividual* e *setPrecoIndividual*, respectivamente).

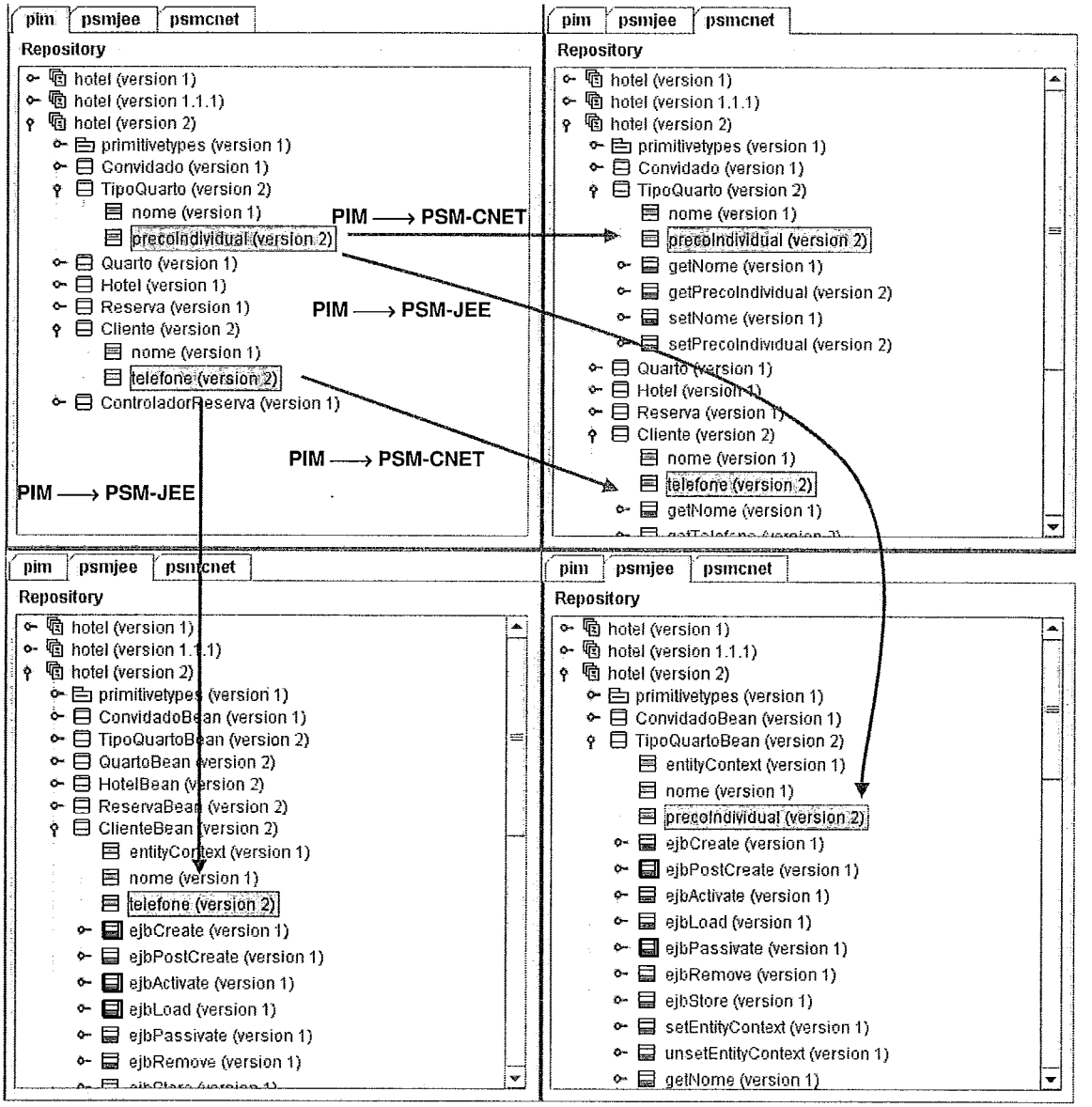

Figura 6.29: PIM e PSMs após a sincronização

6.5.3 Exemplo 3: Modificação do PSM-JEE

Durante a preparação para a geração do código fonte, João percebe a necessidade de um método para verificação da conexão com o servidor de aplicações EJB. Além disso, o engenheiro nota que a classe *ControladorReserva* não possui um método para atualizar as alterações feitas em uma reserva. Apesar deste método estar relacionado com o *PIM*, João não pode esperar por Maria, responsável pelo artefato. Assim, João faz o *check-out* do PSM-JEE e realiza as modificações, que implica na criação do método *checarConexaoEJB*, na classe *ControladorReservaBean*, e o método *alterarReserva*, nesta classe e nas interfaces relacionadas. Em seguida, João importa o

modelo para o cliente do Odyssey-MEC e abre o ModelMarker para inserir os devidos estereótipos nos métodos criados. Além disso, João insere uma etiqueta no método *checarConexao*, para informar que o método não deve ser propagado para os outros modelos. Finalmente, João faz o *check-in* do modelo, gerando as novas versões, como apresentado na Figura 6.30. Neste exemplo, foi realizada uma sincronização direita-esquerda, para atualizar o PIM a partir do PSM-JEE, e uma esquerda-direita, para atualizar o PSM-CNET a partir do PIM.

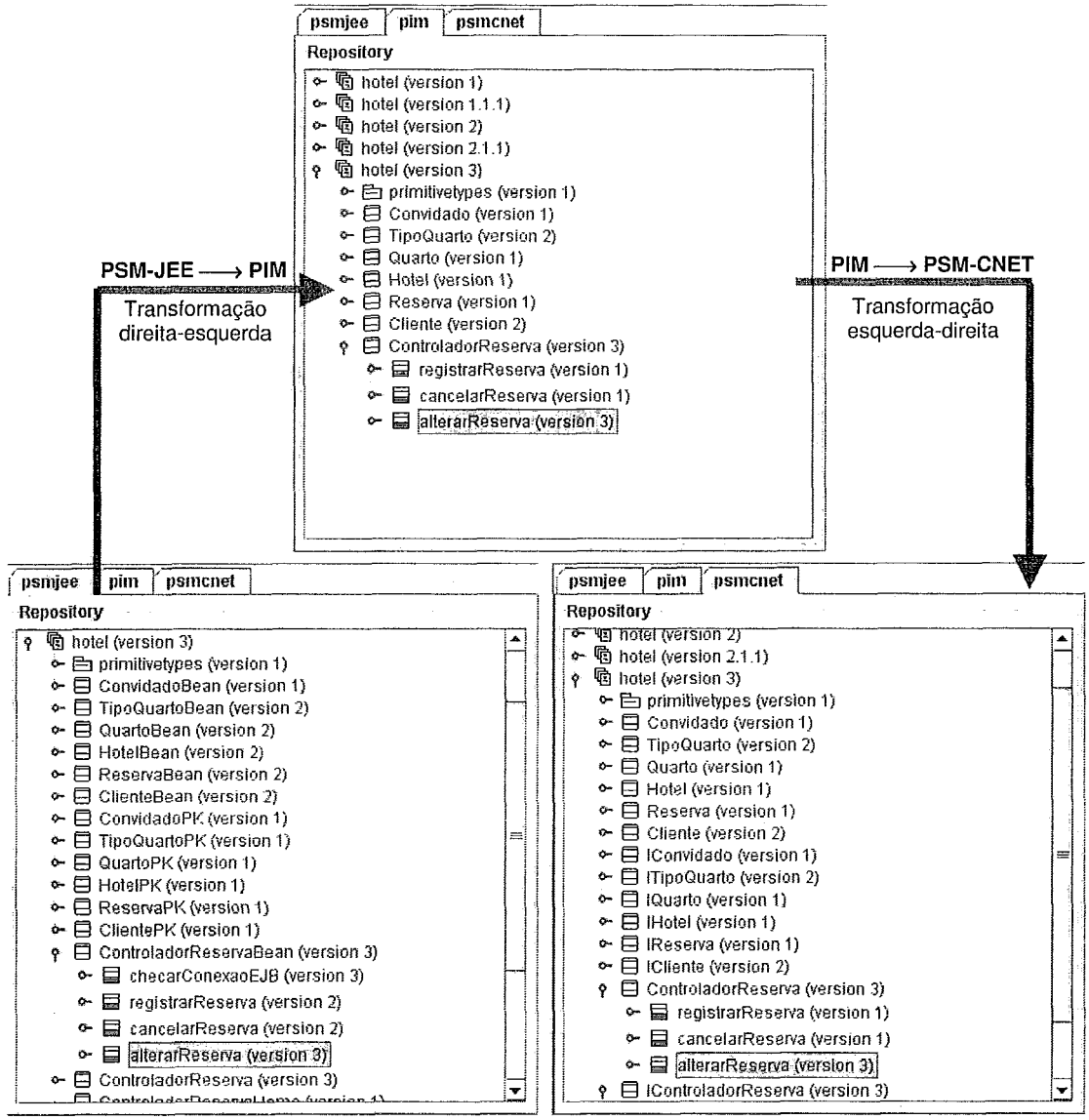

Figura 6.30: PIM e PSMs após alteração do PSM-JEE e sincronização

6.5.4 Exemplo 4: Conflito de Alteração entre Modelos Diferentes

Para realizar novas alterações, Maria e João fazem o *check-out* do PIM e do PSM-JEE, respectivamente. Maria altera o nome do atributo *nome* da classe *Hotel* para *nomeFantasia*. Ao mesmo tempo, João modifica o nome do atributo correspondente na classe *HotelBean* para *razaoSocial*. Maria faz o *check-in* do PIM. Como foi a primeira, o processo de versionamento, transformação e sincronização são executados normalmente, gerando uma nova versão para cada modelo. Logo em seguida, João faz o *check-in* do PSM-JEE. Contudo, ocorre um conflito, decorrente da modificação realizada por Maria. Neste caso, apesar de Maria ter alterado um modelo diferente, o processo de sincronização provocou a alteração do mesmo atributo que João modificou. Assim, quando João fez o *check-in*, a sua alteração conflitou com a modificação realizada pelo Odyssey-MEC (Figura 6.31).

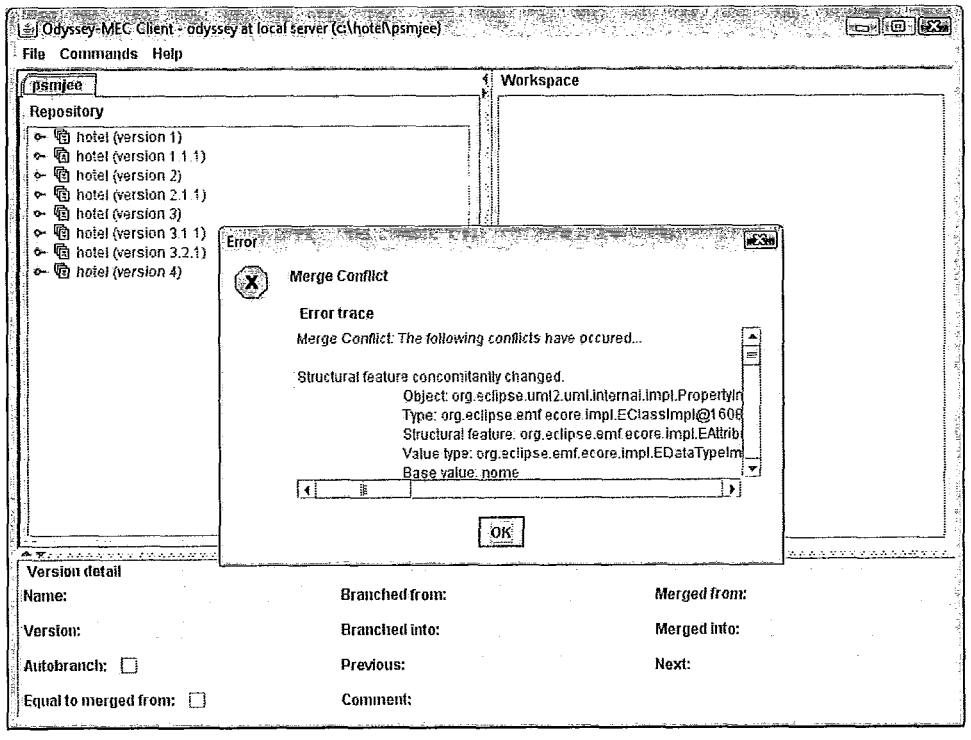

Figura 6.31: Conflito provocado por modelos diferentes

6.6 Considerações Finais

Neste capítulo, foi apresentada a implementação do Odyssey-MEC, uma abordagem para o controle da evolução no tempo e no espaço de modelos inter-relacionados. A partir da ferramenta, é possível realizar a transformação automática de

modelos e independente da iniciativa do desenvolvedor. Quando existem versões anteriores dos modelos gerados a partir de transformações, o Odyssey-MEC realiza a sincronização através de recuperação de informações de versionamento, fazendo com que o componente Odyssey-VCS faça a junção da versão anterior com a nova versão do modelo.

A ferramenta permite o uso de qualquer ferramenta CASE que tenha o recurso de exportação e importação de modelos baseados no padrão XMI 2.1. O modelo deve ser exportado da ferramenta e importado para o cliente do Odyssey-MEC. Após o *check-in*, o processo de controle de evolução é realizado. Para fazer modificações no modelo, o engenheiro de software deve fazer o *check-out,* a exportação do modelo e a importação para a ferramenta CASE.

Para demonstrar a capacidade do Odyssey-MEC em controlar a evolução de modelos inter-relacionados, alguns exemplos foram apresentados, como a modificação de modelos diferentes que implicaram em transformações e sincronizações esquerda-direita, e vice versa. Além disso, foi apresentado um exemplo que demonstra a propagação de alterações de um PSM para o PIM e para os PSMs de outras plataformas. Esta atualização é necessária caso as alterações estejam no nível de abstração do PIM. Foi também demonstrada a situação em que uma alteração está relacionada apenas com o modelo, não devendo ser propagada para os outros.

No próximo capítulo, são apresentadas as conclusões sobre a abordagem, incluindo suas contribuições, limitações e trabalhos futuros.

Capítulo 7 – Conclusões

7.1 Contribuições

O controle da evolução de modelos é um recurso essencial para o Desenvolvimento Dirigido por Modelos (MDD – *Model Driven Development*). Dois aspectos essenciais para alcançar este objetivo são: manter as versões dos modelos à medida que são modificados e garantir a consistência de modelos diferentes que estão inter-relacionados. Estes aspectos caracterizam o controle da evolução de modelos no tempo e no espaço. Contudo, como pôde ser observado no capítulo 3, as abordagens relacionadas com o controle de evolução de modelos estão focadas em apenas uma dessas dimensões. Ao contrário dessas abordagens, o Odyssey-MEC, apresentado nesta dissertação, mantém o controle temporal e espacial dos modelos.

Para realizar o controle da evolução dos modelos, o Odyssey-MEC realiza a transformação e sincronização automática e independente da iniciativa dos engenheiros de software. A execução automática de transformações e sincronizações evita os erros que poderiam ser cometidos caso fossem realizadas manualmente. A independência da iniciativa do desenvolvedor garante que essas operações sejam sempre realizadas. Desse modo, novos modelos são criados quando necessário e todos permanecem consistentes entre si.

Os modelos de um projeto MDD estão inter-relacionados quando um modelo é usado para a geração de outro através de transformação. O fluxo normal é a criação de modelos mais detalhados a partir de modelos menos detalhados e/ou geração de modelos específicos a partir de modelos genéricos. Contudo, é possível que seja necessário realizar o processo inverso. No contexto do MDA, por exemplo, pode ser necessário gerar o PIM a partir de PSM. No caso de sincronização, o PIM pode ter que ser atualizado em função de alterações no PSM. Para resolver esses problemas, o Odyssey-MEC realiza a transformação e sincronização de modelos nas duas direções. Nesse caso, a direção da transformação ou sincronização é determinada pelo modelo-fonte e pela sua posição na transformação, definida no momento da configuração do projeto. Assim, se o modelo recebido pelo Odyssey-MEC estiver configurado como o da esquerda de uma transformação, a direção será da esquerda para a direita, e vice-versa.

Para realizar a sincronização de modelos, o Odyssey-MEC, através do componente de transformação de modelos (Odyssey-MDA), gera as ligações de rastreabilidade necessárias ao processo de sincronização. Essas ligações são mantidas pelo Odyssey-MEC e podem ser usadas com outras finalidades.

A arquitetura cliente-servidor do Odyssey-MEC possibilita o desenvolvimento distribuído. Desse modo, os desenvolvedores podem trabalhar em um projeto MDD de qualquer lugar. Além disso, como a transformação e sincronização são realizadas pelo servidor, as atualizações dos modelos tornam-se disponíveis imediatamente.

O uso da linguagem UML como padrão para a criação de modelos permite a aplicação do Odyssey-MEC a abordagens MDD que usam o MDA como *framework*. Os modelos são persistidos, importados e exportados de acordo como o padrão XMI 2.1. Assim, qualquer ferramenta CASE que importe e exporte os modelos de acordo com esse padrão pode ser usada para a criação e manipulação de modelos cuja evolução deve ser mantida pelo Odyssey-MEC.

7.2 Limitações

Apesar do Odyssey-MEC realizar o controle da evolução de modelos computacionais no tempo e no espaço, a abordagem apresenta algumas limitações, que são apresentadas nas próximas subseções.

7.2.1 Uso de Padrões da OMG

Apesar de um dos requisitos estabelecidos ter sido o uso de padrões definidos pela OMG, a abordagem não utiliza o MOF (OMG, 2006) como meta-metamodelo, nem o QVT (*Query-View-Transformation*) (OMG, 2008) como padrão para a definição de transformações. O MOF não foi usado porque o EMF (BUDINSKY *et al.*, 2003), utilizado como o recurso para representação e manipulação de modelos, é baseado no meta-metamodelo *Ecore*. Contudo, vale a pena ressaltar que o *Ecore* é compatível com o MOF (BUDINSKY *et al.*, 2003). O QVT não foi usado devido a primeira versão oficial ter sido disponibilizada apenas em abril de 2008, inviabilizando a sua utilização na abordagem.

7.2.2 Dependência de Especificação de Transformações

O Odyssey-MEC depende da especificação de transformações do Odyssey-MDA. Desse modo, não é possível usar outras especificações.

7.2.3 Uso de Apenas um Modelo como Entrada

O componente Odyssey-MDA recebe apenas um modelo como entrada para realizar uma transformação. Contudo, é possível que a geração de modelos-alvo dependa de mais de um modelo-fonte, como, por exemplo, um PIM que representa uma solução genérica para o controle de acesso e outro que representa o restante do sistema em desenvolvimento.

7.2.4 Verificação de Consistência Intra-Modelo

Os elementos de um modelo podem estar relacionados com outros elementos de um mesmo modelo. Por exemplo, os métodos *getNome* e *setNome* de uma classe de *Entidade* para a plataforma JEE estão relacionados com o atributo *nome*. Desse modo, a modificação de um desses elementos deve refletir em alterações nos outros elementos. O Odyssey-MEC não realiza o controle desse tipo de evolução.

7.2.5 Propagação Indireta de Transformações entre PSMs

No contexto do MDA, os elementos dos PSMs, normalmente, são gerados a partir dos elementos de um PIM, e vice-versa. Desse modo, os elementos desses modelos estão inter-relacionados. A inclusão, alteração ou exclusão de elementos em um modelo implica na mesma operação nos outros modelos. Contudo, é possível que um PSM tenha elementos específicos que não são propagados para o PIM, como, por exemplo, um método que executa uma operação específica para o PSM. Porém, existe a possibilidade de que o PSM de outra plataforma necessite do mesmo tipo de operação. Assim, apesar de não ser propagada para o PIM, a modificação deveria ser repassada para o outro PSM. Este cenário não é contemplado pelo Opdyssey-MEC. Uma solução para esta situação é definir uma transformação de um PSM para o outro que contemple os elementos específicos.

7.2.6 Geração de Modelos de Comportamento

O Odyssey-MDA gera apenas modelos estruturais baseados em classes e componentes. Desse modo, o Odyssey-MEC não tem como transformar e nem manter a consistência no espaço de modelos comportamentais.

7.2.7 Desempenho

Apesar de não ter sido realizado uma avaliação da abordagem, a execução dos exemplos apresentados no Capítulo 5, baseados em um modelo com sete classes, indicaram que o desempenho do protótipo pode ser baixo para modelos com um grande número de classes.

7.3 Trabalhos Futuros

Este trabalhou foi focado apenas nos modelos computacionais do padrão MDA. Contudo, CIM e o código fonte fazem parte de um projeto MDD que utiliza este *framework*. Além disso, pode ser necessário restringir as alterações realizadas nos modelos. A visualização de como esses modelos evoluem ao longo do tempo também pode fornecer dados importantes para a gerência de configuração. Estes trabalhos futuros, entre outros, são apresentados nas subseções a seguir.

7.3.1 Uso de Regras para Controlar a Evolução dos Modelos

É possível que desenvolvedores façam modificações no modelo-alvo que estão no escopo do modelo-fonte, essas modificações são repassadas para o modelo-fonte durante o processo de sincronização. Contudo, modificações decorrentes de engenharia reversa podem não ser permitidas, de modo a garantir que o processo de desenvolvimento siga o fluxo normal de análise e projeto. Desse modo, o uso de regras de integridade é essencial para determinar o que pode e não pode ser modificado. Assim, caso alguma regra não fosse atendida, as operações de sincronização e versionamento seriam canceladas, garantindo a integridade da seqüência de criação de modelos determinada pelo processo.

7.3.2 Controle da Evolução do CIM

Esta abordagem, inicialmente, propõe o controle da evolução de modelos computacionais (PIM e PSM). Contudo, o controle da evolução do CIM, em sincronia com o PIM e o PSM, tornaria a abordagem mais completa, permitindo o trabalho em um nível ainda mais alto de abstração.

7.3.3 Controle da Evolução de Código Fonte

Apesar dos modelos serem os principais artefatos para o desenvolvimento de software no MDD, o objetivo final é a geração de código fonte. Contudo, esse artefato

pode evoluir independentemente dos modelos. Desse modo, um trabalho futuro é manter a consistência do código fonte com o PSM correspondente.

7.3.4 Controle de Evolução de Mapas de Transformação

Assim como os modelos, mapas de transformação podem ser modificados ao longo do tempo. Manter o controle da evolução desse artefato é essencial para identificar a versão que foi usada para a realização de uma transformação.

7.3.5 Controle de Metamodelos

O Odyssey-MEC possibilita o controle de evolução de modelos baseados no meta-modelo da UML. Contudo, este metamodelo pode evoluir. Além disso, o metamodelo de cada modelo pode ser diferente. Desse modo, outro trabalho futuro é possibilitar o controle da evolução de metamodelos e de modelos baseados em metamodelos diferentes.

7.3.6 Visualização da Evolução dos Modelos Inter-Relacionados

A visualização de modificações permite que engenheiros de software tenham como perceber melhor como o software está evoluindo ao longo do tempo. Um dos benefícios da visualização é permitir a identificação de elementos que possuem um índice mais alto de modificações e como estas modificações estão influenciando a modificação de outros elementos.

Em relação à abordagem proposta, a visualização permitiria acompanhar não somente a evolução do PIM ou do PSM, mas também como um está evoluindo em relação ao outro. Por exemplo, seria mais fácil identificar os tipos de alterações realizadas no PSM que estão provocando alterações no PIM a partir do processo de sincronização.

7.3.7 Avaliação da Abordagem

Apesar dos exemplos apresentados demonstrarem a utilidade da abordagem, é necessário fazer uma avaliação que possibilite verificar o seu uso em situações reais de desenvolvimento. O Odyssey-MEC deve ser avaliado com base em projetos reais e a partir de modelos com grande quantidade de elementos. É também necessário realizar avaliações relacionadas à precisão, desempenho e escalabilidade da abordagem.

Referências Bibliográficas

ACCELERATED-TECHNOLOGY, 2006, "Nucleus BridgePoint". In: http://www.acceleratedtechnology.com/embedded/nuc_modeling.html, accessed in 18/10/2006.

ALANEN, M., 2002, *A Meta Object Facility-Based Model Repository with Version Capabilities, Optimistic Locking and Conflict Resolution*, Master Thesis, Department of Computer Science, Abo Academy University, Turku, Finland.

ALANEN, M., PORRES, I., 2005, "Model Interchange Using OMG Standards". In: *Proceedings of the 31th Euromicro Conference on Software Engineering and Advanced Applications*, pp. 450-458, Porto, Portugal, September, 2005.

ALTMANNINGER, K., 2007, "Models in Conflict - Towards a Semantically Enhannced Version Control System for Models". In: *Proceedings of the Dectoral Symposium at the ACM/IEEE 10th International Conference on Model-Driven Engineering Languiages and Systems*, pp. 293-304, Nashville, USA, September-October/2007.

ALTMANNINGER, K., BERGMAYR, A., SCHWINGER, W., *et al.*, 2007, "Semantically Enhanced Conflict Detection between Model Versions in SMoVer by Example". In: *Proceedings of the International Workshop on Semantic-Based Software Development (SBSD)*, pp. 21-25, Montreal, Canada, October/2007.

ANDROMDA, 2008, "AndroMDA". In: http://galaxy.andromda.org/index.php, accessed in 06/08/2008.

APACHE, 2008, "Apache Tomcat". In: http://jakarta.apache.org/tomcat/index, accessed in 15/03/2008.

APACHE, 2008, "Web Services - Axis". In: http://ws.apache.org/axis/, accessed in 15/03/2008.

BARTELT, C., 2008, "Consistence Preserving Model Merge in Collaborative Development Processes". In: *Proceedings of the International Workshop on Comparison and Versioning of Software Models (CVSM08)*, pp. 13-18, Leipzig, Germany, May/2008.

BERSOFF, E.H., HENDERSON, V., SIEGEL, S., 1980, *Software Configuration Management: An Investment in Product Integrity*, 1 ed., Prentice-Hall.

BEYDEDA, S., BOOK, M., GRUHN, V., 2005, *Model-Driven Software Development*, Spring.

BILLIG, A., BUSSE, S., LEICHER, A., *et al.*, 2004, "Platform Independent Model Transformation Based on Triple". In: *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, v. 78, pp. 493-511, Toronto, Canada.

BIRSAN, D., 2005, "On Plug-Ins and Extensible Architectures", *Queue*, v. 3, n. 2 (03/2005), pp. 40-46.

BOOCH, G., RUMBAUGH, J., JACOBSON, I., 2005, *UML - Guia do Usuário*, 2 ed., Editora Campus.

BOOTH, D., HASS, H., MCCABE, F., *et al.*, 2005, *Web Services Architecture - W3C Working Group Note*, World Wide Web Consortium (W3C).

BROWN, A.W., 2004, "Model Driven Architecture: Principles and Practice ", *Software System Modeling*, v. 3, n. 4, pp. 314-327.

BUDINSKY, F., STEIBERG, D., MERKS, E., *et al.*, 2003, *Eclipse Modeling Framework: A Developer's Guide*, Addison Wesley.

CEDERQVIST, 2005, "Version Management with CVS", *Free Software Foundation, Inc.*

CHEN, F., YANG, H., QIAO, B., *et al.*, 2006, "A Formal Model Driven Approach to Dependable Software Evolution". In: *Proceedings of 30th Annual International Computer Software and Applications Conference - Cover*, v. 1, pp. 205 - 214, Chicago, Illinois, USA, September, 2006.

COLLINS-SUSSMAN, B., FITZPATRICK, B.W., PILATO, C.M., 2004, *Version Control With Subversion*, O'Reilly.

COMPUWARE, "OptimalJ - Model-driven Java Development Tool". In: http://www.compuware.com/products/optimalj/, accessed in 19/10/2007.

CORRÊA, C.K.F., MURTA, L., WERNER, C.M.L., 2008a, "Odyssey-MEC: Model Evolution Control in the Context of Model-Driven Architecture ". In: *Twentienth International Conference on Software Engineering and Knowledge Engineering*, pp. 67-72, Redwood City, CA, USA, July, 2008.

CORRÊA, C.K.F., MURTA, L., WERNER, C.M.L., 2008b, "Odyssey-MEC: Uma Ferramenta para o Controle da Evolução no Contexto do Desenvolvimento

Dirigido por Modelos". In: *XXII Simpósio Brasileiro de Engenharia de Software - XV Sessão de Ferramentas*, pp. 19-24, Campinas - SP, Outrubro/2008.

CZARNECKI, K., HELSEN, S., 2003, "Classification of Model Transformation Approaches". In: *Online Proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, Anaheim, California, USA, October, 2003.

ECLIPSE, 2008, "Eclipse". In: www.eclipse.org, accessed in 10/10/2008.

ECLIPSE, 2008, "EMF-based UML 2.x Metamodel Implementation". In: http://www.eclipse.org/uml2, accessed in 10/04/2008.

ECLIPSE, 2008, "EMF Model Transaction". In: http://www.eclipse.org/modeling/emf/?project=transaction#transaction, accessed in 12/09/2008.

ECLIPSE, 2008, "GEF - Graphical Editing Framework". In: www.eclipse.org/gef, accessed in 10/10/2008.

ELVER, 2008, "Elver Persistency (Eclipse Project as Teneo)". In: http://www.elver.org, accessed in 12/12/2008.

ENGELS, G., KÜSTER, J.M., HECKEL, R., *et al.*, 2002, "Towards Consistency-Preserving Model Evolution ". In: *Proceedings ICSE Workshop on Model Evolution*, pp. 129-132, Florida, USA.

ESTUBLIER, J., 2000, "Software Configuration Management: a Roadmap". In: *Proceedings of 22nd International Conference on Software Engineering, The Future of Software Engineering*, pp. 279-289, Limerick, Ireland, June/2000.

ESTUBLIER, J., GARCIA, S., VEGA, G., 2003, "Defining and Supporting Concurrent Engineering Policies in SCM". In: *Proceedings of the XI International Workshop on Software Configuration Management (SCM 11)*, pp. 1-15, Portland, Oregon, USA, May/2003.

FLATER, D., 2002, "Impact of Model-Driven Standards". In: *Proceedings of the 35th Annual Hawaii International Conference on System Sciences HICSS-35*, pp. 3706 - 3714, Big Island, Hawaii, USA January, 2002.

FONDEMENT, F., SILAGHI, R., 2004, "Defining Model Driven Engineering Processes". In: *3rd Workshop in Software Model Engineering UML 2004 (WISME @ UML 2004)*, Lisbon, Portugal, October, 2004.

FONG, C.K., 2006, "Quick Start Guide to MDA - A Primer to Model-Driven Architecture Using Borland Together Technologies", *Borland*.

FORWARD, A., LETHBRIDGE, T.C., 2008, "Problems and Opportunities for Model-Centric Versus Code-Centric Software Development: A Survey of Software Professionals". In: *Proceedings of the 30th International Conference on Software Engineering*, pp. 27-32, Leipzig, Germany, May/2008.

GÎRBA, T., DUCASSE, S., 2006, "Modeling History to Analyze Software Evolution", *Journal of Software Maintenance: Research and Practice (JSME)*, v. 18, n. 3, pp. 207-236.

GRAY, J., LIN, Y., ZHANG, J., 2006, "Automating Change Evolution in Model-Driven Engineering", *IEEE Computer, Special Issue on Model-Driven Engineering*, v. 39, n. 2 (February/2006), pp. 51-58.

HAYOOD, D., 2008, "MDA: Nice Idea, Shame About the..." In: http://www.theserverside.com/articles/article.tss?l=MDA_Haywood, accessed in 09/05/2008.

IEEE, 2004, *Software Engineering Body of Knowledge (SWEBOK)* IEEE.

IETF, 2008, "RFC 4122 - Universally Unique IDentifier ". In: http://www.ietf.org/rfc/rfc4122.txt, accessed in 10//06/2008.

INTERACTIVE-OBJECTS, 2006, "ArcStyler Overview". In: http://www.interactiveobjects.com/products/arcstyler-overview, accessed in 28/10/2006.

IVKOVIC, I., KONTOGIANNIS, K., 2004, "Tracing Evolution Changes of Software Artifacts through Model Synchronization". In: *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pp. 252-261, Chicago, Illinois, USA, Settember/2004.

JOUAULT, F., KURTEV, I., 2005, "Transforming Models with ATL". In: *Proceedings of the Model Transformation in Practice Workshop at MoDELS*, pp. 128-138, Montego Bay, Jamaica.

KENNEDY-CARTER, 2008, "xUML - Executable UML". In: http://www.kc.com/xuml.php, accessed in 09/06/2006.

KEPPLE, A., WARMER, J., BAST, W., 2002, *MDA Explained: The Model Driven Architecture: Practice and Promice*, Addison-Wesley.

KÖGEL, M., 2008, "Towards Software Configuration Management for Unified Models". In: *Proceedings of of the 2008 International Workshop on Comparison and Versioning of Software Models*, pp. 19-24, Leipzig, Germany, May/2008.

LIN, Y., 2005, "A Model Transformation Approach to Automatic Model Construction and Evolution". In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 448-451, Long Beach, CA, USA, November/2005.

LINEHAN, M.H., 2008, "SBVR Use Cases". In: *Proceedings of The International RuleML Symposium on Rule Interchange and Applications*, v. 5321, pp. 182-196, Orlando - Florida - USA, October, 2008.

MAIA, N.E.N., 2006, *Odyssey-MDA: Uma Abordagem para Transformação de Modelos*, Tese de Mestrado, COPPE/UFRJ, Rio de Janeiro - RJ.

MATHESON, D., FRANCE, R., BIEMAN, J., *et al.*, 2004, "Managed Evolution of a Model Driven Development Approach to Software-based Solutions". In: *Workshop on Best Practices for Model Driven Development*, Vancouver, Canada, October, 2004.

MATULA, M., 2003, *Netbeans Metadata Repository*, NetBeans Community.

MELLOR, S.J., BALCER, M.J., 2002, *Executable Uml: A Foundation for Model-Driven Architecture*, Addison-Wesley.

MELLOR, S.J., SCOTT, K., UHL, A., *et al.*, 2004, *MDA Distilled*, Addison Wesley.

MICROSOFT, "Microsoft .NET Home Page". In: http://microsoft.com/net, accessed in 14/01/2008.

MURTA, L., CORRÊA, C.K.F., PRUDÊNCIO, J.G., *et al.*, 2008, "Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System". In: *Proceeedings of the International Workshop on Comparison and Versioning of Software Models (CVSM08)*, pp. 25-30, Leipzig, Germany, September, 2008.

MURTA, L.G.P., DANTAS, H.L.R., LOPES, L.G.B., *et al.*, 2005, "Odyssey-SCM: An Integrated Software Configuration Management Infrastructure for UML Models", *Science of Computer Programming*, v. 65, n. 3, pp. 249-274.

MYERS, E.W., 1986, "An O(ND) Difference Algorithm and its Variations", *Algorithmica*, v. 1, n. 2, pp. 251-266.

NGUYEN, T.N., MUNSON, E.V., BOYLAND, J.T., 2004, "The Molhado Hypertext Versioning System". In: *Proceedings of the Fifteenth ACM Conference on Hypertext and Hypermedia*, pp. 185 - 194, Santa Cruz, CA, USA.

OLIVEIRA, H., MURTA, L., WERNER, C.M.L., 2005, "Odyssey-VCS: a Flexible Version Control System for UML Model Elements". In: *International Workshop*

on Software Configuration Management (SCM-12) pp. 1-16, Lisbon, Portugal, September, 2005.

OLIVEIRA, H.L.R.D., 2005, *Odyssey-VCS: Uma Abordagem de Controle de Versões para Elementos da UML*, Tese de Mestrado, COPPE/UFRJ, Rio de Janeiro, RJ – Brasil.

OMG, 2002, *CORBA Component Model, Version 3.8*, Object Management Group.

OMG, 2003a, "Common Warehouse Metamodel (CWM) Specification, Version 1.1", *Object Management Group.*

OMG, 2003b, "MDA Guide Version 1.0.1", *Object Management Group.*

OMG, 2005a, "Meta Object Facility (MOF) Specification - Version 1.4.1", *Object Management Group*, 12/07/2007.

OMG, 2005b, *XML Metadata Interchange (XMI) Specification. Version 2.0*, Object Management Group.

OMG, 2006, "Meta Object Facility (MOF) Core Specification - Version 2", *Object management Group*, 09/06/2008.

OMG, 2007, *Unified Modeling Language (UML) Infrastructure Specification. Version 2.1.2*, Object Management Group.

OMG, 2008, "MOF 2.0 Query/View/Transformation Specification", *Object management Group.*

PORRES, I., 2002, *A Toolkit for Manipulating Models*, Turku Centre for Computer Science 441.

PRESSMAN, R.S., 2002, *Engenharia de Software*, 5 ed., Rio de Janeiro, Mc Graw Hill.

SANGWAN, R., BASS, M., MULLICK, N., *et al.*, 2007, *Global Software Development Handbook*, Auerbach Publications.

SEBESTA, R.W., 2005, *Concepts of Programming Languages*, 7 ed., Addison Wesley.

SENDALL, S., KOZACZYNSKI, W., 2003, "Model Transformation - The Heart and Soul of Model-Driven Development", *IEEE Software*, v. 20, n. 5 (September/October, 2003), pp. 42-45.

SENDALL, S., KÜSTER, J., 2004, "Taming Model Round-Trip Engineering". In: *Workshop on Best Practices for Model-Driven Software Development*, Vancouver, Canada, October/2004.

SMOVER, 2008, "SMOVER - A Semantically Enhanced Version Control System for Models". In: http://smover.tk.uni-linz.ac.at/7/publications.php, accessed in 23/06/2008.

SOLEY, R., 2000, *Model Driven Architecture*, Object Management Group, OMG document omg/00-05-05.

SPANOUDAKIS, G., ZISMAN, A., 2001, "Inconsistency Management in Software Engineering: Survey and Open Research Issues", *World Scientific Pub. Co.*, n. 1, pp. 329-380.

SPARX, 2008, "MDA Overview", *Sparxs Systems*.

SRIPLAKICH, P., 2007, *ModelBus - An Open and Distributed Environment for Model Driven Engineering*, Ph.D. Thesis, University Pierre and Marie Curie, Paris, France.

SRIPLAKICH, P., BRANC, X., GERVAIS, M.-P., 2008, "Collaborative Software Engineering on Large-Scale Models: Requirements and Experience in ModelBus". In: *Proceedings of the 2008 ACM Symposium on Applied Computing*, pp. 674-681, Fortaleza, CE, Brazil, March/2008.

SUN, 2007, "The Java EE 5 Tutorial". In: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html, accessed in 13/12/2007.

SUN, 2008, "Sun's Java Tutorial". In: http://java.sun.com/docs/books/tutorial/, accessed in 20/01/2008.

TIAN, J., 2005, *Software Quality Engineering*, Hoboken, New Jersey - USA, John Wiley & Sons.

TRUYEN, F., 2005, "Implementing Model Driven Architecture Using Entreprice Architect - Version 1.1", *Sparx Systems*, July/2008.

XIONG, Y., LIU, D., HU, Z., *et al.*, 2007, "Towards Automatic Model Synchronization from Model Transformations". In: *Proceedings of the 22th IEEE/ACM International Conference on Automated Software Engineering*, pp. 164-173, Atlanta, Georgia, USA.

XML-RPC, 2008, "XML-RPC Specification". In: http://www.xmlrpc.com, accessed in 13/12/2008.