Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Pós-Graduação em Ciência da Computação

FEATURES-ORIENTED MODEL-DRIVEN ARCHITECTURE: UMA ABORDAGEM PARA MDD

Fábio Paulo Basso

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: Prof. Dr. Toacy Cavalcante Oliveira

Porto Alegre, março de 2006

Av. Ipiranga, 6681 - Prédio 16
CEP 90619-900 - Porto Alegre - RS - Brasil
http://www.pucrs.br/biblioteca

Fone: 0 (xx) 51 3320-3544
FAX: 0 (xx) 51 3320-3548
e-mail:bceadm@pucrs.br

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Reconhecido pelo Parecer No. 930/98.C.N.E. Homologação Publicada no D.O.U. de 30/12/98.

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "***Features-Oriented Model-Driven Architecture: Uma Abordagem para MDD***", apresentada por Fábio Paulo Basso, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas de Informação, aprovada em 31/03/2006 pela Comissão Examinadora:

Prof. Dr. Toacy Cavalcante de Oliveira – Orientador — PPGCC/PUCRS

Profa. Dra. Karin Becker – PPGCC/PUCRS

Prof. Dr. Leandro Buss Becker – UFSC

Homologada em 28/02/07, conforme Ata No. 03 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

AV. IPIRANGA, 6681 - PRÉDIO 16/SALA 106
90619-900 - PORTO ALEGRE -RS -BRASIL
TELEFONE: 55 51 33203611 FAX 55 51 3320.3621
http: //www.pucrs.br/inf/pos e-mail: ppgcc@inf.pucrs.br

Dedico essa obra aos meus pais Irani e
Maria e também a minha noiva Raquel

Dedicatória

AGRADECIMENTOS

A realização do presente trabalho contou com o apoio e a participação de várias pessoas. Como sábios mestres, estas pessoas souberam indicar boas soluções para problemas encontrados durante o mestrado. A todos aqui mencionados gostaria de reportar meus sinceros agradecimentos.

Orientador Toacy Cavalcante Oliveira, pela amizade, apoio e encorajamento dados durante a elaboração dessa obra e por sua disposição para discussões, que propiciaram boas contribuições no presente trabalho. Um agradecimento especial pela paciência e pelo empenho com que esta pessoa orientou meu trabalho.

Professor Leandro Buss Becker, por contribuir com essa pesquisa. Um agradecimento também pela amizade, o encorajamento e pelo apoio dados desde o início do mestrado.

Professora Karin Becker, pelo incentivo dado durante uma difícil fase do mestrado que enfrentei.

Aos meus pais Irani e Maria e minhas irmãs Cíntia e Paula, pelo patrocínio, apoio, confiança, exemplo e encorajamento dados durante o mestrado.

À Raquel Mainardi Pillat, por toda a dedicação, paciência e incentivo dados durante todo este árduo processo e por ajudar na revisão desse trabalho.

A HP (Hewlett Packard) por manter a bolsa que financiou o mestrado.

RESUMO

O desenvolvimento de software dirigido por modelos, com a MDA, requer o refinamento de modelos de sistemas, inicialmente especificados em alto nível e sem características de plataformas, em modelos dependentes de plataformas. A Model Driven Architecture (MDA) apresenta uma técnica de modelagem voltada para arquiteturas, em que é necessário aplicar mapeamentos e transformações em modelos de sistemas para plataformas.

Esse trabalho apresenta a abordagem FOMDA (*Features-Oriented Model-Driven Architecture*), que permite a desenvolvedores especificar modelos e gerenciar transformações adotando a técnica MDA. A abordagem FOMDA combina Modelos de Features e MDA em um ambiente onde desenvolvedores são habilitados para representar características através de *features*, mapeando-as para transformações que geram a aplicação final. Este trabalho também apresenta um estudo de caso, que utiliza a abordagem FOMDA para auxiliar no desenvolvimento de um sistema de tempo real embarcado.

ABSTRACT

The model driven software development with MDA requires the transformation of system models, initially specified in high-level and platform independent, into platform dependent models. The Model Driven Architecture (MDA) presents an architectural guided modeling technique, using mappings and transformations that must be applied in the system models according system platforms.

This work presents the FOMDA (*Features-Oriented Model-Driven Architecture*) approach, to help developers specifying models and managing transformations using the MDA technique. The FOMDA approach combines Features Model and MDA in an environment where developers can represent system characteristics with features and map them to transformation code that is responsible to generate the final application. This work also presents a case study using FOMDA to assist the development on an embedded real-time system.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

CIM	*Computation Independent Model*
COTS	*Component Of The Shelf*
DCT	*Diagrama de Composição de Transformações*
ED	*Engenharia de Domínio*
ES	*Engenharia de Software*
FM	*Features Model*
FOMDA	*Features-Oriented Model-Driven Architecture*
GP	*Generative Programming*
JVM	*Java Virtual Machine*
M2C	*Model-to-Code*
M2M	*Model-to-Model*
MDA	*Model Driven Architecture*
MDD	*Model Driven Development*
MOF	*Meta Object Facilities*
MVC	*Model, View Controller*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
PIM	*Platform Independent Model*
PLA	*Product Line Architecture*
PSM	*Platform Specific Mode*
QoS	*Quality of Service*
QVT	*Query, View and Transformation*
QVTP	*QVT-Partners*
RDL	*Reuse Description Language*
RF	*Requisito Funcional*
RNF	*Requisito Não Funcional*
RT	*Real Time*
RTSJ	*Java Real-Time System*
SiEs	*Sistema Embarcados*
SoC	*System on-Chip*
SETR	*Sistema Embarcado de Tempo Real*

SPT	*Schedulability, Performance and Time UML Profile*
TMD	*Transformação de Manipulação Direta de modelos*
UML	*Unified Modeling Language*
UML-FI	*UML for Features Instantiation*
USB	*Universal Serial Bus*
XMI	*XML Metadata Interchange*
XML	*Extensible Markup Language*
XSLT	*XSL Transformation Language*

SUMÁRIO

1 Introdução

O Desenvolvimento de software Dirigido por Modelos (MDD) requer o refinamento de modelos de sistemas, inicialmente especificados em alto nível e sem características de plataformas, em modelos dependentes de plataformas. A Model Driven Architecture (MDA) é uma abordagem para MDD em que é sugerido que os modelos de sistemas sejam especificados em três visões (MDA 2006): CIM (Computation Independent Model), que especifica modelos independentes de computação e de alto nível; PIM (Platform Independent Model), que especifica modelos independentes de plataforma que também são representações de alto nível; e PSM (Platform Specific Model), que especifica modelos dependentes de plataforma que são representações de baixo nível.

A MDA define que os modelos podem ser refinados de uma visão para outra utilizando um conjunto de mapeamentos de modelos de sistemas para transformações. Boas (2005) também define que é necessário organizar estas transformações para possibilitar que um modelo especificado em alto nível (PIM) seja transformado para um outro dependente de plataforma (PSM). No entanto, não é especificado na MDA como é possível administrar e organizar mapeamentos, transformações e modelos de sistemas para estas visões. O presente trabalho propõe uma abordagem para organizar mapeamentos de modelos para transformações que refinam modelos de sistemas na visão PIM para modelos na visão PSM.

O aspecto chave na MDA é permitir que o desenvolvimento de um sistema seja decomposto em níveis de abstração das plataformas que serão utilizadas para desenvolver um sistema. O termo plataforma é usado nesse documento para representar características de sistemas, como arquiteturas, tecnologias, serviços, hardware, ferramentas, ambientes de desenvolvimento, etc. Além disso, é possível aplicar um conjunto de transformações nos modelos do sistema e gerar novos modelos e código específico para determinadas plataformas. De acordo com Selic (2005), o termo plataforma pode ser compreendido como "qualquer conjunto de mecanismos de hardware ou software que habilitam a execução de aplicações de software". No entanto, a MDA não define um formalismo para especificar modelos em níveis de independência das características de plataformas.

Um formalismo utilizado por projetistas para modelar e especificar regras para combinação de plataformas é o Modelo de Features (FM) (Tekinerdogan et al. 2004, Czarnecki 1998). Ele é composto por características (representando requisitos), informações textuais, como regras de composição, e *rationales,* que são informações detalhadas a respeito de uma característica (Czarnecki e Eisenecker 2004, Basso, Oliveira e Becker 2006). A idéia é definir características (funcionais, arquiteturais, tecnológicas, etc.) de um domínio particular e, com base nas relações entre elas, usá-las para estabelecer diferentes configurações entre plataformas.

Esse trabalho apresenta a abordagem FOMDA (*Features-Oriented Model Driven Architecture*), que permite que desenvolvedores especifiquem modelos e gerenciem transformações adotando a técnica MDA. A abordagem FOMDA combina Modelos de Features e MDA em um ambiente onde desenvolvedores são habilitados a identificar mapeamentos e transformações de modelos para plataformas alvo usadas no desenvolvimento de um sistema.

As plataformas são especificadas como uma combinação de características em um Modelo de Features (que pode ser visto na Figura 1 (a) e (b)). Essas características são mapeadas para transformações, como pode ser visto na Figura 1 (a) e (c). A seleção das características do Modelo de Features (FM) determina o conjunto de transformações que devem ser realizadas no refinamento de um modelo especificado em alto nível (PIM), para modelos de baixo nível (PSMs). Isto é possível de ser visto na Figura 1 (b) e (c), em que as características selecionadas F1 e F3 identificam as transformações T1 e T3. Muitas transformações podem ser utilizadas para refinar um PIM em PSMs. Então é necessário ordenar as características selecionadas do FM em um *workflow*. Isto é mostrado na Figura 1 (b) e (d).

Esse trabalho também apresenta um estudo de caso. Com o objetivo de comprovar que a abordagem FOMDA pode ser utilizada para auxiliar a identificar os mapeamentos e as transformações dos modelos de um sistema. Esse estudo de caso demonstra como é possível refinar um modelo em alto nível (PIM) para um modelo em baixo nível (PSM). O PIM é um modelo que contém requisitos de um sistema para controle de cadeira de rodas e o PSM é um modelo o qual pode ser utilizado para gerar código para o desenvolvimento de um sistema de tempo real embarcado.

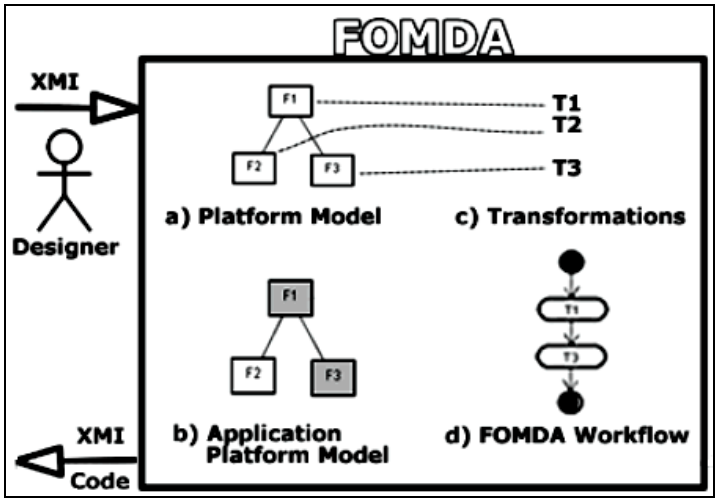


Figura 1 - Abordagem Features-Oriented Model-Driven Architecture

1.1 Motivação

As abordagens existentes para MDA, como em Deeltra et al. (2003), Oliveira et al. (2004) e Almeida et al. (2004), não tratam adequadamente o desenvolvimento de sistemas para múltiplas plataformas. Estas abordagens estão centradas em prover soluções para transformar modelos abstratos em modelos concretos e usar estes últimos para gerar código. No entanto, elas não possibilitam gerar código para múltiplas plataformas, porque plataformas não auxiliam nas transformações dos modelos. Diante disso, o presente trabalho propõe uma nova abordagem para MDD, que permite o desenvolvimento de sistemas, a partir de modelos, para múltiplas plataformas.

A Object Management Group (OMG) é o grupo responsável em padronizar a MDA. Neste âmbito, este grupo lança chamadas de propostas para abordagens com foco em MDD. Uma de suas chamadas é referida como Query, Views and Transformations (QVT) e relata às contribuições relacionadas com consultas, visões e transformações de modelos na MDA. Este trabalho apresenta uma contribuição para QVT, que motivou o presente trabalho a propor uma abordagem para documentar e visualizar mapeamentos e transformações de modelos. Essa abordagem possibilita que consultas, visões e transformações sejam especificadas no processo de mapeamento e transformação de modelos.

Na MDA, modelos abstratos são refinados para modelos concretos com o auxílio de transformações. As transformações são realizadas por transformadores de modelos. Os transformadores especificados em ferramentas como Jamda (Jamda 2006) e UMT (UMT 2006) são estáticos. Um transformador é estático quando não é possível modificar a sua transformação. Existem propostas em que os transformadores são dinâmicos (Willink 2003). Em transformadores dinâmicos é possível modificar as transformações que ele realiza, assim como criar novos transformadores a partir de transformadores pré-fabricados. Como as tecnologias disponíveis para desenvolver um sistema mudam com freqüência (Graaf, Lormans e Toetenel 2003, Basso, Cavalcante e Becker 2006), a maioria dos transformadores de modelos precisa ser reescritos para efetuar novas transformações. Isto ocorre porque estes transformadores são desenvolvidos para aplicar transformações para uma tecnologia específica. Logo, eles não são reutilizáveis, se existirem mudanças nas tecnologias utilizadas pelos sistemas.

Algumas vantagens para o reuso de transformadores de modelos são: a) independência de plataformas; b) a possibilidade de compor novos transformadores, em tempo de execução, a partir de outros já desenvolvidos; c) a possibilidade de modificar transformações de acordo com as mudanças das plataformas alvo. Em nenhuma das abordagens estudadas, como UMT, Jamda e RDL (Oliveira 2001), os transformadores implementam todos estes aspectos. No presente trabalho, todos os aspectos mencionados foram desenvolvidos. Para tornar isto possível, a organização de transformações foi dividida em quatro etapas de abstração. Em cada nível é possível especificar transformadores em níveis altos de abstração em relação às características não funcionais de sistemas.

1.2 Objetivos

Esse trabalho tem como principal objetivo descrever uma nova solução para MDD (denominada FOMDA), utilizando a MDA e o Modelo de Features. Como objetivos específicos tem-se:

- Apresentar uma abordagem para gerar o código de um sistema para múltiplas plataformas alvo com base em um modelo especificado com a UML em alto nível;

- Apresentar uma solução para transformar um PIM em um PSM;
- Apresentar uma solução para reutilização e composição de transformações;
- Apresentar uma ferramenta que auxilia os projetistas na transformação de um PIM para um ou mais PSMs;
- Apresentar um estudo de caso voltado para o domínio de sistemas embarcados. Com este estudo, pretende-se atestar que os objetivos anteriores foram atingidos no presente trabalho.

1.3 Contribuições

O presente trabalho apresenta uma abordagem para desenvolvimento de software dirigido por modelos denominada FOMDA. Esta abordagem oferece recursos para a representação dos modelos de um sistema nas visões da MDA e, além disso, utiliza o Modelo de Features para estabelecer a combinação de plataformas que precisam ser utilizadas no desenvolvimento de uma aplicação. Para isso, esta abordagem divide o mapeamento e a transformação de modelos (explicados na Seção 2.4.4) em quatro níveis de abstração. Cada nível deve ser utilizado para definir os mapeamentos e transformações de um modelo de sistema.

Um protótipo denominado FOMDA Toolkit foi desenvolvido para validar a abordagem FOMDA. O protótipo tem por objetivo: a) utilizar o Modelo de Features para representar as características não funcionais de um domínio de sistemas; b) permitir a configuração das plataformas alvo, utilizadas para desenvolver um sistema, com base na seleção das características do Modelo de Features; c) utilizar as plataformas alvo para efetuar e compor transformações nos modelos do sistema; d) possibilitar a organização das transformações em quatro níveis de abstração; e) utilizar os padrões definidos pela OMG como UML (Boch 1998), MOF (MOF 2003) e XMI (XMI 2003).

Para validar a abordagem, este trabalho traz um estudo de caso para o desenvolvimento de um Sistema de Tempo Real Embarcado (STRE). Esse estudo de caso é apresentado no Capítulo 4 e detalha como transformar um modelo genérico, especificado em

alto nível (especificado com a UML e sem características de plataformas), para um modelo específico de uma plataforma alvo (identificada em um FM).

Um artigo foi submetido e aceito no congresso (IEEE/ International Symposium on Object-Oriented Real-Time Distributed Computing- ISORC 2006) e foi publicado em Abril de 2006 (Basso, Cavalcante e Becker 2006). Este artigo apresentou a abordagem FOMDA como uma solução para o desenvolvimento dirigido a modelos e um exemplo do uso desta abordagem para o desenvolvimento de SETRs (o artigo consta no Anexo I deste documento).

1.4 Organização

O restante do trabalho está organizado como segue. O Capítulo 2 apresenta as principais bases teóricas sobre a análise de requisitos para sistemas embarcados. O Capítulo 3 apresenta as abordagens centradas na geração de código com base em modelos. O Capítulo 4 apresenta a Model Driven Architecture. O Capítulo 5 apresenta os trabalhos relacionados. O Capítulo 6 descreve a abordagem FOMDA. O Capítulo 7 descreve as etapas de transformação da FOMDA. O Capítulo 8 apresenta o protótipo de ferramenta FOMDA Toolkit. O Capítulo 9 apresenta um estudo de caso para o desenvolvimento de sistemas embarcados de tempo real, utilizando a abordagem FOMDA. As conclusões são apresentadas no Capítulo 10.

2 Análise de Requisitos para Sistemas Embarcados

A fase de análise de requisitos da Engenharia de Software (ES) tem como função descobrir as características relacionadas a um sistema. Ela inclui um processo de desenvolvimento voltado para o domínio de sistemas e inclui os aspectos relacionados à arquitetura e características de funcionalidades do sistema. Segundo Douglass "o conjunto de objetos externos significantes e suas interações com o sistema, formam a base para a análise de requisitos do sistema" (Douglass 1998). Para todo e qualquer objeto que exerça uma ação no sistema ou que seja acionado por ele, este faz parte da análise de requisitos.

Sistemas Embarcados (SiEs) são sistemas que, em sua grande maioria, possuem pouca disponibilidade de recursos para efetuar o processamento dos dados de entrada (Wagner e Carro 2003). Os recursos podem ser entendidos como sendo físicos (de hardware computacional e de espaço físico disponível), e lógicos (softwares leves com o mínimo de funcionalidades e restrito ao hardware). SiEs podem ser encontrados em máquinas de lavar, aparelhos de gravação e reprodução de DVDs, simuladores de vôo, aviões, robôs, brinquedos, cafeteiras, etc. Em um sistema embarcado a análise de requisitos da Engenharia de Software também pode ser empregada (Graaf e Lormans 2003). Entretanto, nesta classe de sistemas alguns requisitos adicionais devem ser considerados, como, por exemplo, o limite de consumo de energia sem perda de desempenho, a baixa disponibilidade de memória, a necessidade de segurança e confiabilidade, e o curto tempo de projeto.

Geórgia e Jaelson (2003) dividem requisitos de sistemas em Requisitos Funcionais (RF) e Requisitos Não Funcionais (NFR). RFs capturam o comportamento do sistema em termos de serviços, tarefas ou funções que o mesmo necessita fazer. Nesta categoria enquadram-se requisitos que descrevem os objetivos do sistema. Por exemplo, em um sistema de cadeira de rodas, o objetivo é mover a cadeira de rodas utilizando algum controle. Para cumprir com este objetivo, os requisitos funcionais da cadeira de rodas são: as direções para as quais ela pode ir, quantos motores são necessários para mover a cadeira, quais dispositivos controlam a mesma, etc. Requisitos funcionais geralmente são aqueles que significam entrada, processamento e saída de dados, controles, exceções e entidades.

Geórgia e Jaelson definem ainda requisitos não funcionais (RNF). Estes são ditos como aqueles que impõem restrições no produto sendo desenvolvido, no processo de desenvolvimento do sistema, ou que especificam restrições externas que o produto ou processo necessita conhecer. Exemplos de RNFs do mesmo sistema de cadeira de rodas são: o tempo mínimo para que o dispositivo de freio seja acionado no momento em que a ordem de parar a cadeira for dada, o tempo mínimo de comunicação entre o dispositivo de controle e os motores, o requisito de tolerância a falhas que define que o dispositivo de freio e a comunicação com ele não pode falhar, etc.

2.1 Características que Devem ser Consideradas Durante a Análise de Requisitos de Sistemas Embarcados

Algumas características devem direcionar quais requisitos fazem parte do sistema. Como SiEs são de natureza e funcionalidades distintas, muitas características podem ser relevantes no momento da identificação dos requisitos do sistema e outros não, no entanto é necessário ter conhecimento de quais aspectos são característicos destes sistemas. Com base nestas características, muitas configurações podem ser estabelecidas. Estas características englobam:

1. **Necessidade**: quais as funcionalidades o sistema deve oferecer;
2. **Performance**: em quanto tempo uma determinada tarefa deve ser cumprida;
3. **Força**: qual a potência que um determinado aparelho tem para a execução de um determinado trabalho;
4. **Memória**: quanto de memória é necessário para que o sistema cumpra com suas funcionalidades;
5. **Paralelismo**: quais tarefas podem ou devem ser paralelizadas para que a execução de uma tarefa mais global seja completada em um tempo menor do que se fosse executada seqüencialmente;
6. **Fluxo de dados**: a quantidade de dados que o sistema processa e/ou envia para suas entradas e saídas (E/S);
7. **Segurança**: quem pode ter acesso ao sistema;

8. **Tolerância a falhas**: quais áreas do sistema, ou o sistema por completo, não podem falhar em hipótese alguma;
9. **Sensores**: quais dispositivos de sensoriamento o sistema tem a disposição e qual o objetivo de cada um;
10. **Acionadores**: quais dispositivos disparam alguma tarefa do sistema ou acionam algum dispositivo externo;
11. **Comunicação com o usuário**: qual o tipo de comunicação o sistema utiliza para enviar uma mensagem ao usuário;
12. **Temporizadores**: utilizados para disparar a execução de uma tarefa. Deve-se também saber, para o uso apropriado de temporizadores, se a execução de uma tarefa será periódica (ocorre de tempos em tempos determinados), episódica (quando algum evento acontecer, neste caso ela pode necessitar esperar algum tempo e disparar a execução, disparar uma execução logo que uma mensagem chegar, ou disparar mais de uma execução);
13. **Sistema operacional**: que tipo de sistema operacional deve ser utilizado. É necessário estabelecer para qual sistema operacional o sistema em desenvolvimento poderá funcionar. Em sistemas embarcados há ainda a necessidade de especificar a utilização de um sistema operacional de tempo real, ou um sistema operacional específico para sistema embarcado;
14. **Linguagem de programação**: A escolha do sistema operacional implica na escolha da linguagem de programação. Linguagens de programação como a Java não têm o seu código compilado, mas sim interpretado, necessitando que seja instalada uma JVM (*Java Virtual Machine*) específica do sistema operacional e do hardware. Uma JVM interpreta o código para que funcione naquela determinada arquitetura. Já aplicações desenvolvidas em linguagens compiladas devem ser compiladas para a arquitetura a qual se destinam e no sistema operacional definido;
15. **Disponibilidade de espaço**: em que objeto físico este sistema é empregado. Qual a disponibilidade de espaço neste objeto para a implantação do sistema. O espaço pode implicar em:
    a. Componentes mais caros que os tradicionais, conhecidos como SoC (*System on-Chip*). SoC são sistemas completos em um só chip, contendo processadores, memórias, dentre outros componentes de um sistema computacional. São indicados para serem empregados em objetos que tenham restrição de espaço físico;

b. Componentes de uso comercial, mais conhecidos como *Component Of The Shelf (COTS)*, que são baratos mas ocupam um espaço maior que um SoC pois não foram projetados para espaços pequenos. COTS são componentes utilizados em computadores tradicionais, como placa mãe, memória, etc., que são organizados para aumentar a capacidade de processamento e armazenamento de uma máquina tradicional para satisfazer as necessidades do sistema em desenvolvimento. São indicados para serem empregados em objetos onde a restrição de espaço não é um problema.

16. **Custo com o desenvolvimento do sistema**: quais são os gastos com análise e desenvolvimento do sistema;
17. **Custo de mercado**: com base em todos os aspectos mencionados, qual é o custo total do sistema no momento da venda no mercado. Este aspecto direciona, muitas vezes, os aspectos anteriores, definindo a qualidade de alguns deles e até mesmo a escolha pela não utilização.

2.2 Importância das Características para Domínios de Sistemas Embarcados

A escolha da relevância de uma característica depende do domínio do sistema. No entanto, muitas delas são necessárias em sistemas que são partes do domínio. Com exceção da característica de necessidade, as outras são não funcionais, o que significa que podem estar presentes em muitas aplicações do domínio de sistemas embarcados. Além disso, NFRs freqüentemente mudam conforme evoluem as arquiteturas de hardware. A característica da necessidade pode ser dividida em muitos requisitos funcionais (RF). Então, existem também subdivisões do domínio de sistemas embarcados, em que cada subdivisão especifica funcionalidades para sistemas semelhantes.

Uma subdivisão de características funcionais determina aspectos de um domínio de aplicações. Por exemplo, um sistema embutido em uma cadeira de rodas, que rastreia o solo em busca de obstáculos que possam colidir com ela arranja um conjunto de funcionalidades, algumas obrigatórias e outras opcionais para cada aplicação que venha a ser desenvolvida.

Características obrigatórias podem ser: controlar movimento, acionar trava, etc. Características opcionais podem ser: detectar automaticamente um obstáculo, tomar decisão de trajetória, etc. Então, pode-se organizar um domínio não somente de acordo com o tipo de sistema, como no caso domínio de SiEs e domínio de sistemas comerciais, mas também de acordo com aplicações semelhantes como domínio de aplicações para cadeira de rodas.

Cada aplicação para o domínio de sistemas de cadeira de rodas pode ao mesmo tempo especificar a seguintes necessidades: desempenho para processar as informações, restrições de tempo que definem o tempo máximo para que um aviso de colisão chegue até o usuário, restrições de espaço físico optando-se por um SoC ao invés de um COTS, restrições de força de cada motor da cadeira de rodas, memória suficiente para armazenar as imagens capturadas, etc. Já para um sistema que monitora os freios da cadeira de rodas as características relevantes podem ser outros, como: tolerância a falhas, sensores e acionadores, não há a necessidade de desempenho como para o exemplo anterior (se a resposta não chegar num intervalo de tempo máximo, a cadeira pode colidir).

Características de desempenho e de restrições temporais estão estreitamente relacionadas porque quando for necessário desempenho, os requisitos temporais devem ser avaliados. Tais características são definidas como aspectos de qualidade de serviço QoS. Aspectos de custo são um pouco mais complexos, pois somente podem ser determinados após a definição da necessidade que, por sua vez, determina a disponibilidade de espaço, o desempenho, fluxo de dados, memória, etc, que determinam o custo total do equipamento, efetuando uma relação cíclica entre o custo final, a necessidade, os requisitos desta necessidade e o custo destes requisitos. Assim, com base nestas características, os requisitos da aplicação podem ser identificados e a análise de requisitos pode ser feita.

2.3 Etapas de Análise de Requisitos de Sistemas Embarcados

A especificação de requisitos de sistemas embarcados envolve muitas camadas (Graaf e Lormans 2003), em cada camada, um nível de detalhe adicional do sistema é especificado. No exemplo do domínio de sistema de cadeira de rodas, dois sistemas que possuem RF e RNF

distintos foram identificados, um que rastreia o solo em busca de obstáculos e outro que monitora o dispositivo de freio, cada um deles como um subsistema de um sistema maior que é a cadeira de rodas. Um processo identificado por Graaf e Lormans (2003) define em qual etapa cada requisito é analisado. Estas etapas podem ser vistas na Figura 2. A primeira etapa define os requisitos do sistema em geral, ou seja, requisitos funcionais e não funcionais. Nesta etapa uma arquitetura geral é estabelecida. No exemplo da cadeira de rodas, esta arquitetura seria uma visão geral do sistema, ou seja, a cadeira de rodas necessita de motores, de um subsistema que controle esta cadeira, de um subsistema de rastreia o solo em busca de colisões, de um subsistema que monitore o dispositivo de freio, dentre outros, onde o subsistema que monitora o dispositivo de freio deve obedecer a restrições temporais e de tolerância à falhas, enquanto que o subsistema rastreador de obstáculos deve obedecer a restrições de tempo.

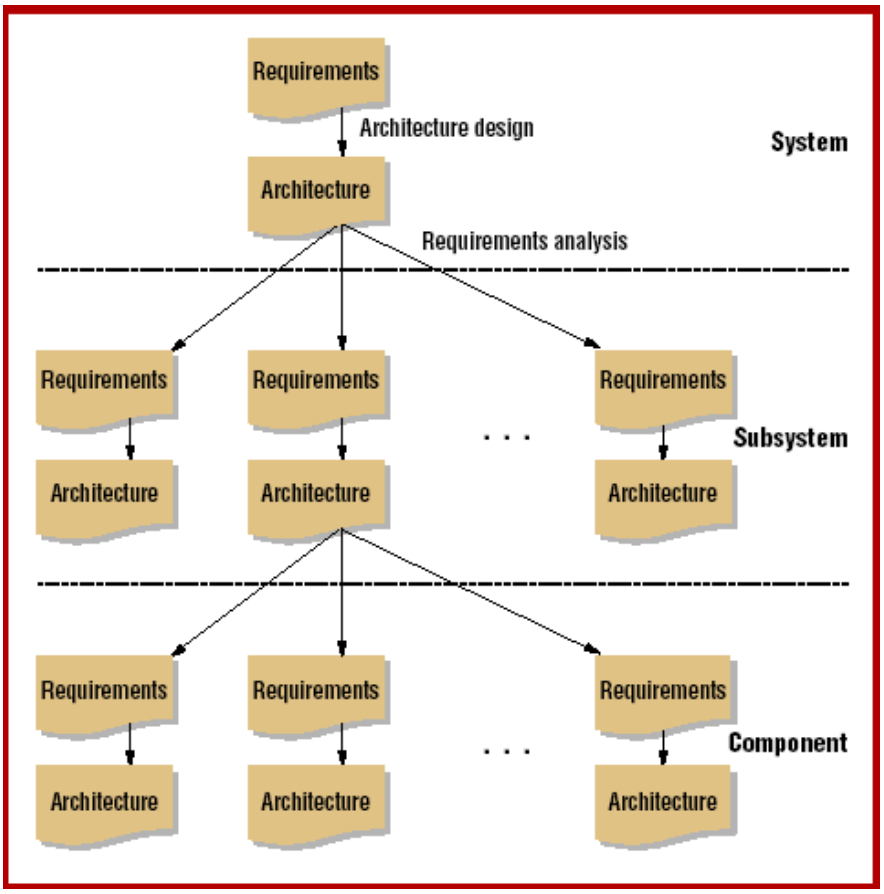


Figura 2 - Camadas de Engenharia de Requisitos (Graaf e Lormans 2003)

A segunda etapa do processo utilizado pelos mesmos autores é identificar os requisitos dos subsistemas, se houverem. Com base nas arquiteturas identificadas no processo anterior, cada arquitetura é separada e seus requisitos funcionais e não funcionais são detalhados. O subsistema que monitora o dispositivo de freio, por exemplo, possui requisitos funcionais que podem ser classificados como detectar falha no dispositivo, corrigir falha, informar ao usuário a falha ocorrida, acionar trava de emergência, etc., e requisitos não funcionais como a trava de emergência que deve ser aciona em tempo real logo que for detectada uma falha no dispositivo de freio especificam a maneira como cada funcionalidade deve ser cumprida pelo sistema. Estas informações formam um conjunto de características do domínio de aplicações para sistemas de cadeira de rodas e características particulares do sistema sendo analisado.

A terceira e última etapa do processo de desenvolvimento de software para sistemas embarcados de Graaf e Lormans é o detalhamento dos subsistemas em nível de componentes. Novamente requisitos são melhores detalhados e finalmente componentes específicos da arquitetura são definidos. Para detectar falha no dispositivo de freio, por exemplo, é necessário utilizar um sensor que detecte que o dispositivo está falhando, assim um componente da sub-arquitetura detectar falha no dispositivo foi encontrada. Para acionar a trava de emergência, é necessário de um componente acionador, que aciona a trava de emergência assim que um problema com o dispositivo de freios for detectado.

Como a restrição da necessidade de acionar a trava de emergência é em tempo real, mas basta enviar um sinal para a trava de emergência, não há a necessidade de um alto processamento nem tampouco de muita memória, uma memória que apenas registre o erro ocorrido é suficiente. No entanto, para comunicar outros subsistemas, como o caso em que seja necessário comunicar os motores e o sistema de detecção de colisão para pararem o funcionamento, há a necessidade de um dispositivo de comunicação rápido como, por exemplo, uma porta USB (*Universal Serial Bus*) que possibilita que o tráfego de informações flua mais rápido que portas seriais. Há ainda a necessidade da programação de *device drivers* específicos para operar cada componente. Estes processos assimilam-se com o ditado “separar para conquistar”, o subsistema é separado em subsistemas, que são separados em outros subsistemas até que se chegue á uma definição de componentes de hardware.

Brisolara (2004) define outra forma de encontrar os requisitos do sistema:

> "A especificação dos requisitos consiste na definição de quatro elementos principais, que são: (i) funcionalidade desejada pelo usuário; (ii) requisitos de Qualidade de Serviço (QoS), tais como: desempenho, restrições temporais, etc; (iii) restrições de custo (área, consumo de potência, etc.); (iv) estrutura do domínio do problema" (Brisolara 2004).

É possível então estruturar os aspectos relevantes do sistema em um dos quatro elementos principais da especificação de requisitos. Esta proposta define uma estruturação das características relevantes do sistema adequadamente, pois facilita a especificação dos seus requisitos e possíveis objetos.

O primeiro dos quatro elementos principais é caracterizado por encontrar as características relevantes nas funcionalidades desejadas pelo usuário. Nesta etapa são avaliadas as tarefas que o sistema deve executar, os objetivos do sistema, os possíveis módulos de operação e interface entre o usuário e o sistema. Aspectos temporais devem ser avaliados no segundo elemento denominado requisitos de qualidade de serviço QoS, como o tempo para execução de uma tarefa, o intervalo de tempo de resposta que um dispositivo leva para acionar outro dispositivo ou software.

As características relativas ao custeio do sistema devem ser avaliadas no terceiro elemento, denominado restrições de custo. Neste elemento são avaliados, de acordo com o sistema, o espaço físico disponível para abrigar o conjunto de software e hardware, os dispositivos (micro-controladores, COTS, as arquiteturas de hardware embutidas no sistema (número de processadores, memória, barramento) e os softwares (sistema operacional de tempo real, linguagem de programação orientada a objetos ou estruturada).

Por fim, na estrutura do domínio do problema, o sistema deve ser enquadrado em sistemas críticos, quando requerem segurança a falhas, requerendo tolerância a altas temperaturas, pressão, falta de energia, e sistemas leves, os quais não lidam com situações extremas.

2.4 O Projeto do Sistema Depois da Especificação dos Requisitos

As maneiras de estruturar a especificação de requisitos, abordadas por Brisolara (2004) e por Graaf e Lormas (2003), são apropriadas para a descoberta dos requisitos do sistema. No entanto, existe um detalhe não abordado nessas maneiras: o projeto do sistema após a especificação de requisitos. Segundo Graaf e Lormans (2003), requisitos capturam o que o sistema deve fazer, enquanto que o projeto mostra como construir o sistema. Na prática, a identificação dos requisitos do sistema é feita de forma adequada, porém o problema surge no momento de projetar o sistema colocando no projeto os requisitos identificados. Há a necessidade de mostrar à equipe de desenvolvimento os requisitos não funcionais que devem ser colocados no sistema, para tanto eles devem ser especificados no projeto, o que não acontece na prática de desenvolvimento de sistemas embarcados. Este é um problema específico de projeto de software onde os requisitos não funcionais são importantes, como projeto de software para SiEs.

Disciplinas de engenharia de software auxiliam na solução destes problemas, introduzindo notações para o projeto de sistemas, em especial para o projeto de sistemas orientados. A programação orientada a objetos é uma metodologia que objetiva descrever as relações do sistema, ou RFs, em que cada objeto do sistema é uma unidade que pode se comunicar com outros objetos e requisitar a execução de uma determinada funcionalidade. Cada RF em uma análise orientada a objetos é uma funcionalidade de um objeto. A notação que mais teve destaque e aceitação para a análise orientada a objetos é a UML (Boch et al 1999). Esta notação oferece maneiras de modelar os RNFs e RF de um sistema de forma adequada, visto que ao final do projeto, o objetivo é descrever a estruturação e relação entre os objetos em um diagrama de classes.

Para suprir a necessidade de especificação de RNFs na fase de projeto, empresas do setor de sistemas embarcados têm utilizado apenas a especificação dos requisitos em linguagem natural, processando-os em um editor de textos simples (Graaf e Lormans 2003). Esta atitude acaba sendo um ponto negativo no projeto destes sistemas, visto que as empresas continuamente reutilizam de partes de projetos anteriores, e fica difícil a organização de requisitos de forma textual, o que acarreta em formas de reuso de projetos não muito

apropriadas. Muitos dos RNFs descritos de forma textual, não são encontrados quando, no momento de reutilizar de projetos já prontos, necessitam ser reaplicados em um novo projeto. A UML oferece um conjunto de Perfis, que compreendem estereótipos, *tagged values* e restrições, que podem ser adicionadas aos requisitos do projeto, especificando como eles devem ser tratados pelo sistema. Como o uso da UML e dos Perfis que ela oferece, é possível especificar muitos dos RNFs identificados nesta Seção.

A definição de RNFs em projetos de sistemas é, portanto, um ponto positivo na modelagem de sistemas embarcados. No entanto, após esta definição, existe uma diferença muito grande entre o código final de uma aplicação e o código que é gerado a partir do modelo do sistema especificado na fase de projeto. Um desafio atual para projetistas deste tipo de sistema é utilizar os NFRs especificados nos modelos de sistemas para gerar código. Isto possibilita a geração de um código mais parecido com o código final das aplicações. Algumas propostas que viabilizam isso são detalhadas na próxima Seção.

3 Geração de Código a Partir de Modelos de Sistemas

Algumas abordagens para desenvolvimento de software embarcado propõem a especificação dos NFRs em modelos de sistemas com o objetivo de gerar código para aplicações. Becker et al. (2002) exemplifica como um modelo que contém requisitos não funcionais é mapeado para código específico de uma API Java denominada RTSJ (Java Real-Time System). Selic (2005) apresenta alguns conceitos que viabilizam o sucesso no desenvolvimento de software embarcado tendo como propósito gerar código para múltiplas plataformas. Estas abordagens tem por objetivo demonstrar como utilizar os NFRs especificados nos modelos de sistema e gerar código fonte para um conjunto de plataformas alvo.

A crescente disponibilidade de diferentes plataformas tem sido um fator determinante para o sucesso no desenvolvimento de software (Selic 2005). Esta disponibilidade permite aos desenvolvedores estabelecerem diferentes configurações entre plataformas. Uma configuração determina as características das plataformas que uma aplicação demanda e que devem estar especificadas no projeto dessa aplicação. Um desafio para o desenvolvimento de software embarcado é utilizar o RNFs, especialmente os que identificam características de plataformas, para gerar código para aplicações. Este capítulo apresenta alguns conceitos que podem contribuir para superar este desafio.

3.1 A Necessidade de Sistemas de Software Independentes de Plataformas

O desenvolvimento de software é um ofício que exige bastante conhecimento das tecnologias disponíveis para a construção de sistemas. Neste ofício, um dos requisitos para uma boa administração da informação e sucesso do desenvolvimento de aplicações é conhecer as plataformas que possibilitam a construção de software, suas restrições e as possibilidades existentes para efetuar a comunicação entre elas.

Tekinerdogan (2004) apresenta exemplos das plataformas disponíveis para construção de software comercial. Nesses exemplos são expostas algumas das plataformas existentes para desenvolvimento de software, informando qual o vendedor, o sistema operacional que pode rodar a plataforma, o tipo de arquitetura utilizada, a linguagem de programação e os serviços da arquitetura.

Tekinerdogan argumenta que existem algumas plataformas que são dependentes. Por exemplo, se a plataforma escolhida no quesito vendedor for Microsoft, necessariamente ou muito provavelmente as sub-plataformas precisam ser da família dos produtos Microsoft, como a linguagem de programação C#, lógica de negócio deve ser .NET, o sistema operacional deve ser Windows, etc. Pode haver uma combinação entre plataformas livres e plataformas Microsoft, no entanto cuidado deve ser tomado ao efetuar este tipo de combinação, como utilizar algumas características ou sub-plataformas Windows, efetuando integração com um sistema UNIX usando CORBA.

A ubiqüidade de plataformas possibilita que um sistema possa ser desenvolvido levando-se em conta muitas de suas características não funcionais. No entanto, a troca freqüente das plataformas usadas para implantar um sistema é um problema visto pelos projetistas de sistemas, porque isto implica no pouco reuso dos artefatos do software deste sistema. Softwares desenvolvidos para uma determinada plataforma são difíceis de serem reutilizados em outra (Basso 2004b). Um exemplo típico de sistemas em que são freqüentes as trocas de plataformas são os embarcados

3.2 Desenvolvimento Dirigido por Modelos (MDD)

Uma das possíveis soluções para resolver o problema do reuso de artefatos de software em decorrência da troca freqüente das plataformas utilizadas para implantar um sistema é utilizar um Desenvolvimento de Software Dirigido por Modelos. Frankel (Frankel 2004) informa que o paradigma MDD é diferenciado da linha de produção da engenharia de software clássica, porque viabiliza um ágil desenvolvimento de software, utilizando a MDA

(explicada na Seção 4). Além disso, o paradigma MDD também possibilita o reuso dos artefatos de um sistema, mesmo quando houver a troca de plataformas.

O MDD adiciona uma prática de "modelagem voltada para arquiteturas". Por exemplo, um projetista pode refinar um modelo em alto nível em um modelo específico de uma arquitetura. Se esta arquitetura é Java e é composta por arquiteturas como EJB e Struts, o modelo refinado pelo projetista deve conter características de EJB e de Struts. Isto pode ser realizado pelo projetista para usar o modelo refinado e gerar um código mais detalhado do que o código que seria gerado utilizando o modelo representado em alto nível (modelo não refinado para arquitetura). Esta prática pode ser aplicada entre muitas das etapas definidas por um processo de desenvolvimento e possibilita a geração de artefatos de software com maiores detalhes do que em práticas tradicionais. Portanto, o uso do MDD por projetistas auxilia no processo de desenvolvimento, reduzindo o tempo necessário para desenvolver sistemas (Greenfield e Short 2003).

Uma das vantagens do MDD é possibilitar a geração de códigos de melhor qualidade e em maior quantidade do que usando práticas tradicionais de desenvolvimento de software (Greenfield e Short 2003). A geração de Código é realizada com o auxílio de ferramentas CASE. Greenfield e Short (2003) informam que ferramentas para geração automatizada de código disponíveis no mercado pouco auxiliaram, até a data de sua publicação, no desenvolvimento de código voltado para as plataformas usadas no desenvolvimento de um sistema (Greenfield e Short 2003). Até o momento, a maioria das ferramentas CASE pouco tem contribuído para a geração de código levando em conta RNFs de sistema, em especial os requisitos que representam características de plataformas. Há algumas exceções, contudo, para a afirmação anterior, como a ferramenta OptimalJ (OptimalJ 2006), que usa os conceitos da MDA para gerar código para sistemas. No entanto, a maioria das ferramentas CASE produzem código ineficiente (Greenfield e Short 2003). Devido a isso, o processo de desenvolvimento de software, em especial para sistemas embarcados, tem obtido pouco proveito de ferramentas CASE (Graaf, Lormans e Toetenel 2003).

No MDD, ao invés de utilizar apenas uma única transformação de um modelo UML diretamente para código, pode-se efetuar mais de uma transformação deste mesmo modelo. A razão disto é que, em cada novo modelo que é gerado, seja possível especificar um detalhe que defina um produto final mais fiel aos requisitos. A Figura 3 mostra algumas das maneiras

utilizadas até o momento para transformar um modelo em código e também efetuar a engenharia reversa, que é transformar código em modelo.

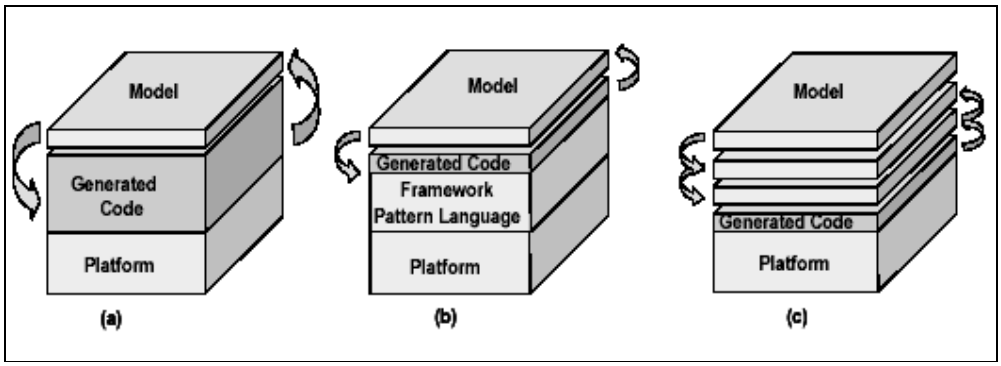


Figura 3 - Ferramentas CASE, do Modelo ao Código (Greenfield e Short 2003)

As primeiras ferramentas CASE seguiam a estrutura mostrada na Figura 3(a) (Greenfield e Short 2003). Nesta estrutura, um modelo é convertido em código, utilizando uma ferramenta CASE de uma plataforma específica, e a engenharia reversa converte o código em modelo. Já na Figura 3(b), foram acrescentados os conceitos de *framework* e padrões de linguagem, pois no modelo são representados requisitos para sistemas de um mesmo domínio. A Figura 3(c) apresenta uma possível representação para ferramentas CASE, que possibilita aplicar um desenvolvimento dirigido a modelos como a MDA.

A Figura 3(b) apresenta a visão adotada pela Engenharia de Domínio (ED). Nesta visão, um *framework* é especificado (um modelo, contendo elementos UML, representando os requisitos de sistemas de um mesmo domínio), oferecendo conjuntos de requisitos que podem ser utilizados por muitos sistemas que possuem requisitos semelhantes. A seleção dos requisitos para cada sistema é realizada utilizando linguagens para seleção dos elementos de *frameworks*, como RDL (Oliveria et al. 2004).

A terceira e mais recente das visões das ferramentas CASE é projetar o modelo do sistema uma única vez e permitir que este modelo sofra sucessivas transformações. Isto é realizado até que os modelos transformados ofereçam uma correta especificação do sistema, mapeado para determinadas plataformas. Por fim, estes modelos são transformados em artefatos de software (Figura 3(c)). Essa última visão representa o esquema de funcionamento da MDA, discutida na Seção 2.4.

3.2.1 Análise das Abordagens para Ferramentas CASE

Uma análise de abordagens para ferramentas CASE foi realizada em (Basso 2004(b)) com o objetivo de refinar modelos em alto nível para modelos de baixo nível e dependentes de plataforma. Nesta análise, foi identificado que as ferramentas CASE que seguem o modelo da Figura 3 (a) não são capazes de gerar código que obedeça a restrições como sincronização, tempo de resposta, dentre outros requisitos não funcionais. Isto ocorre porque elas não levam em consideração a especificação de RNFs, como características de tempo real, para a geração de código. Ferramentas CASE que seguem o modelo da Figura 3 (b), como a encontrada em (Oliveira 2004), facilitam o reuso de modelos, o desenvolvimento e a geração de código de qualidade, mas ainda assim não podem gerar códigos eficientes, como é desejável em aplicações para sistemas embarcados e também comerciais. O terceiro modelo, da Figura 3(c), é mais sugestivo e surge como uma solução para aplicar os RNFs (especificados no modelo do sistema) em código. Em cada modelo pode-se especificar restrições específicas para a plataforma escolhida para desenvolver o sistema, detalhando-as em suas sub-plataformas.

Para realizarem um desenvolvimento dirigido por modelos as ferramentas CASE que tem este propósito podem utilizar contribuições relacionadas com o termo *Generative Programming*, ou simplesmente Programação Gerativa. A próxima seção apresenta algumas abordagens relacionadas com este termo.

3.3 Programação Gerativa

Generative Programming (GP) é uma denominação para algumas abordagens relacionadas com o desenvolvimento de sistemas, que utilizam algum processo de automatização para geração de código. Esse capítulo apresenta algumas abordagens para GP, que tem como foco a MDD.

3.3.1 Engenharia de Domínio

Engenharia de Domínio (ED) é um processo de desenvolvimento de software que visa reutilizar modelos de sistema de mesmo domínio e permitir que estes sistemas sejam desenvolvidos com agilidade (Czarnecki 1998). As técnicas tradicionais de projeto orientado a objetos focam no projeto de um software visando apenas o sistema que necessita ser construído. O processo adotado pela ED avalia requisitos, não somente do sistema que necessita ser desenvolvido, mas também do domínio o qual pertence esta aplicação (Czarnecki 1998, Czarnecki e Eisnecker 2004).

Para Czarnecki (1998), a avaliação e documentação das características do domínio são realizadas em quatro componentes: a) definição do domínio, onde exemplos de aplicações do domínio são expostos, bem como regras genéricas para inclusão e exclusão no domínio; b) léxico do domínio; c) modelos conceituais, representados por diagramas de objeto, de interação, de estados (relacionados com a UML) ou diagramas de entidade-relacionamento e fluxo de dados (relacionados com a análise estruturada); d) Modelo de *Features* (FM - *Features Model*), que explicita em um diagrama a composição das características do domínio.

Um FM pode contemplar RNFs utilizados por sistemas embarcados, documentando-os no projeto das aplicações do mesmo domínio. Estes requisitos podem ser representados como características do FM. Segundo Griss (2001), uma característica pode ser: uma seleção entre requisitos opcionais ou alternativos; uma funcionalidade; um requisito não funcional (como performance, hardware e arquitetura de implementação).

3.3.2 Modelo de Features

É possível utilizar o FM para especificar plataformas. Tekinerdogan et al. (2004) propuseram uma abordagem para especificar regras para seleção de plataformas, em formato textual no FM. As regras definidas por estes autores são muito úteis para definir uma correta configuração das características do FM que podem compor um sistema. O FM pode satisfazer a necessidade de sistemas (como os embarcados) pela representação das arquiteturas, tecnologias e etc., utilizadas para o desenvolvimento destes sistemas (Basso, Oliveira e Becker 2006).

O Modelo de Features também é conhecido como *Diagrama de Features* (Czarnecki 1998). Existem muitos formalismos para especificar um FM (Czarnecki 1998, Griss 2001, Garlan 1996, Tekinerdogan et al. 2004, Kang et al. 1999). O formalismo adotado nesse trabalho é o proposto por Czarnecki (Czarnecki 1998, Czarnecki e Eisnecker 2004), apresentado na Figura 4, que mostra os tipos de relacionamentos que podem ser estabelecidos entre as características no MF. Como o FM é hierárquico, sendo interpretado muitas vezes como uma estrutura em árvore (Griss 2001, Kang et al. 1999) e em outras abordagens como um grafo (Czarnecki 1998, Oliveira et al. 2004, Basso, Oliveira e Becker 2006), a composição das características é representada de forma que as de nível hierárquico superior são compostas pelas características do nível inferior. Por exemplo, na Figura 4(a), a característica *"Feature"* pode ser composta pela *"FeatureA"*. Na Figura 4(f), a característica *"Feature"* é composta pela(s) característica(s) *"FeatureG"* e/ou *"FeatureH"*.

A representação do Modelo de Features de Czarnecki define as seguintes relações entre características (representadas na Figura 4): (a) relação opcional, usada quando a característica relacionada é opcional; (b) relação obrigatória, usada quando a característica relacionada é obrigatória; (c) relação de dependência, usada quando uma característica é dependente de outra característica definida no FM. A relação de dependência pode ser representada ou pela regra de composição *requires 'feature'*, a qual é uma informação textual adicionada à característica relacionada, ou graficamente pela relação de dependência, como mostrada na Figura 5; (d) relação obrigatória mutuamente exclusiva, usada quando o projetista precisa escolher uma característica de um conjunto de características; (e) relação opcional mutuamente exclusiva, usada quando o projetista pode escolher uma característica de um conjunto de características; (f) relação "ou", usada quando pelo menos uma característica de um conjunto de características é necessária; (g) relação "zero ou mais", usada quando um conjunto de características são opcionais (é similar à situação representada na Figura 4(f), mas os círculos não são preenchidos).

O Diagrama de Features apresentado na Figura 5 pode expressar funcionalidades obrigatórias do sistema, funcionalidades mutuamente exclusivas e funcionalidades opcionais. Além disso, o Modelo de Features é muito adequado também para modelar plataformas (Tekinerdogan et al. 2004), porque ele possibilita combinar as características definidas no modelo e, assim, configurar uma ou mais plataformas. A Figura 5 apresenta um FM para especificar características para aplicações do domínio de cadeira de rodas.

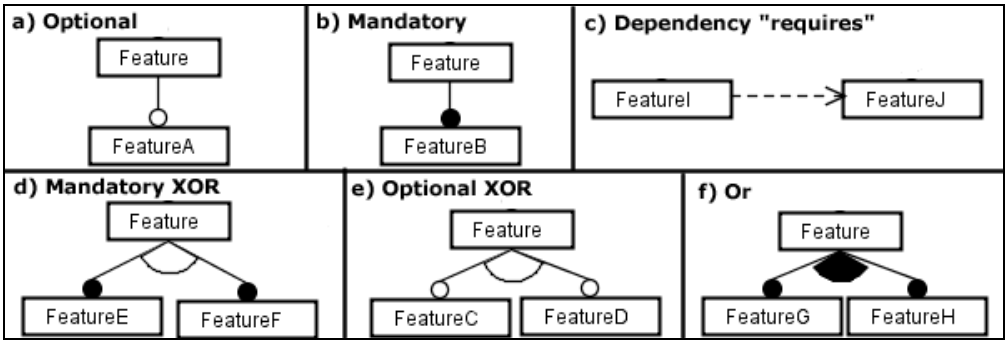


Figura 4 - Relações do FM Definidas por Czarnecki (1998)

É possível especificar regras de dependência entre características no Diagrama de Features. Usando extensões propostas por Tekinerdogan et al. (Tekinerdogan et al. 2004) ou por Riebisch et al. (Riebisch et al. 2002), é possível especificar algumas regras para composição de características, que são dependentes, mas que não são possíveis de serem representadas utilizando apenas a sintaxe do Modelo de Features definido por Czarnecki. Tais extensões são especificações textuais aplicadas em cada característica. Para Oliveira et al. (Oliveria et al. 2004) e Basso et al. (Basso, Oliveira e Becker 2006), a seta que simboliza dependência no diagrama UML é utilizada para indicar relacionamentos de dependência.

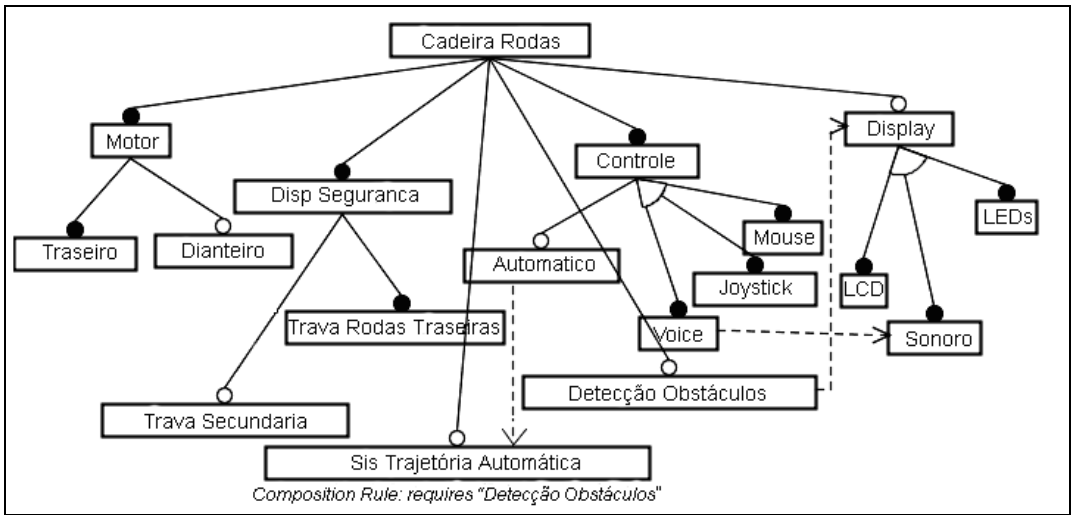


Figura 5 - FM para o Domínio de Aplicações de Cadeira de Rodas

A Figura 5 apresenta características para o domínio de sistemas para cadeira de rodas. De acordo com as características definidas e as relações entre elas especificadas no modelo, é obrigatório, para cada sistema do domínio um motor traseiro, um dispositivo de segurança

para trava de rodas traseiras da cadeira e um controle que deve ser ou por voz, ou por *mouse* ou por *joystick*. O uso de um motor dianteiro, uma trava secundária e um controle automático são opcionais. Um *display*, um sistema de detecção de obstáculos e um sistema de trajetória automática são opcionais para o desenvolvimento de um sistema de cadeiras de rodas. Porém, caso o controle por voz tenha sido selecionado, então um *display* sonoro torna-se obrigatório para a cadeira de rodas.

Se um sistema de detecção de obstáculos for selecionado no domínio, então é obrigatório o uso de um *display* para informar ao usuário da cadeira de rodas os obstáculos. Além disso, caso um controle automático tenha sido selecionado no domínio de cadeira de rodas, então é obrigatório que a cadeira de rodas tenha um sistema automático para decidir a trajetória a ser tomada pela cadeira de rodas. Neste caso, a *composition rule "requires 'Detecção Obstáculos'"*, definida na característica *"Sist. Trajetória Automática"* do FM define que: ao ser selecionada, então a detecção de obstáculos é necessária também. As regras para configuração das características são definidas pelas relações entre elas e por suas *compositions rules*. De acordo com as regras definidas no FM, muitas configurações podem ser estabelecidas entre as características.

Uma correta especificação de um FM para um domínio de aplicações é abordada pela disciplina de Arquiteturas para Linha de Produtos (PLA - *Product Line Architectures*).

3.3.3 Arquiteturas para Linha de Produto

Um conjunto de produtos que compartilham um conjunto de requisitos, onde cada produto além dos requisitos compartilhados possui também seus próprios requisitos, são ditos como representantes de uma Arquitetura para Linha de Produto (PLA) (Griss 2001). Uma PLA endereça um domínio de aplicações (Griss 2001). Logo, um domínio de aplicações contempla aplicações semelhantes. Assim, ED e PLA são assuntos interligados, pois uma PLA configura muitos produtos identificados pela ED. Enquanto a segunda avalia o contexto geral das aplicações, buscando a decomposição de características de sistemas, a primeira avalia especificações das aplicações deste contexto geral, buscando o agrupamento delas.

Combinações das características de sistemas (identificadas em uma PLA) determinam o que uma aplicação do domínio definido pela PLA deve conter. Cada aplicação pode estender estas características de duas maneiras: 1) adicionando novas funcionalidades; 2) definindo novos requisitos não funcionais. No primeiro caso, abordagens para *Generative Programming* oferecem recursos para a geração de elementos em modelos de sistemas, para cada configuração da linha de produção. No segundo caso, não existem abordagens que possibilitam gerar artefatos de um sistema que atendam aos requisitos não funcionais, como a geração de código para múltiplas plataformas de desenvolvimento, de maneira automatizada ou semi-automatizada. Por este motivo, tipicamente, as abordagens para PLA limitam-se à geração dos artefatos de um sistema para as configurações funcionais especificadas na linha de produção.

Os engenheiros de PLA utilizam o FM mais a UML para especificar os requisitos do sistema (Oliveira et al. 2004). O Modelo de Features oferece aos engenheiros de PLA uma maneira organizada de representar as variações na linha de produção. A UML é uma notação que oferece a possibilidade de especificar RFs e RNFs de sistemas, que se tornou um padrão para a especificação destes. O FM especifica RFs e RNFs de maneira mais geral, enquanto os elementos da UML possibilitam especificá-los detalhadamente. A diferença entre projetos convencionais e de PLA é que os engenheiros da linha de produção estabelecem uma relação entre as características do FM e os requisitos representados em UML e utilizam isto para a geração automatizada ou semi-automatizada de código. A análise de requisitos utilizando a UML é realizada de acordo com o processo de desenvolvimento adotado, sendo o mais recomendado o processo baseado em Engenharia de Domínio. Neste último, os requisitos identificados na análise são extraídos de um conjunto de aplicações semelhantes.

Ao especificar um FM, o engenheiro de PLA analisa um conjunto de funcionalidades (em geral, classes especificadas com a notação UML) que compõem cada característica. Tipicamente, durante a análise, o engenheiro de PLA: especifica uma relação entre as classes/funcionalidades do domínio e cada característica do FM; marca os elementos das classes como obrigatórios ou opcionais (usando estereótipos e *tagged values*); define um conjunto de funcionalidades para cada característica do FM, que podem ser automaticamente

selecionadas de acordo com a instanciação[1] do Modelo de Features; define um *framework* de funcionalidades para o domínio.

3.3.4 Instanciação do Modelo de Features

O Modelo de Features é utilizado como um mapa que descreve uma linha de produção. Ele oferece componentes para seleção e personalização, que permitem acrescentar as características desejadas em uma determinada aplicação (Griss 2001). No entanto, esse modelo é conceitual e seus elementos (ou características) não podem diretamente gerar código (Oliveira et al. 2004). O Modelo de Features se encaixa muito bem no desenvolvimento de um *framework*, pois expressa características que descrevem o domínio da aplicação. Estas características, transformadas em elementos do Diagrama de Classes, podem ser instanciadas e reutilizadas (Oliveira 2001). Após a conclusão da Análise de Domínio e especificação das características no FM, o processo adotado na Engenharia de Domínio é a Instanciação do Modelo de Features (Kang et al. 1998).

A instanciação do FM (conhecida em PLA como instanciação para linha de produto) é realizada selecionando características do FM. Essa seleção é mostrada na Figura 6. As características em destaque na figura representam uma instanciação do FM. Sempre que uma nova aplicação for desenvolvida com base no domínio especificado pela DE, a instanciação do FM deve ser realizada. Portanto, as características marcadas na Figura 6 representam os requisitos de uma determinada aplicação.

[1] O termo instanciação do Modelo de Features é usado para representar a seleção das características do diagrama pelo projetista de uma aplicação.

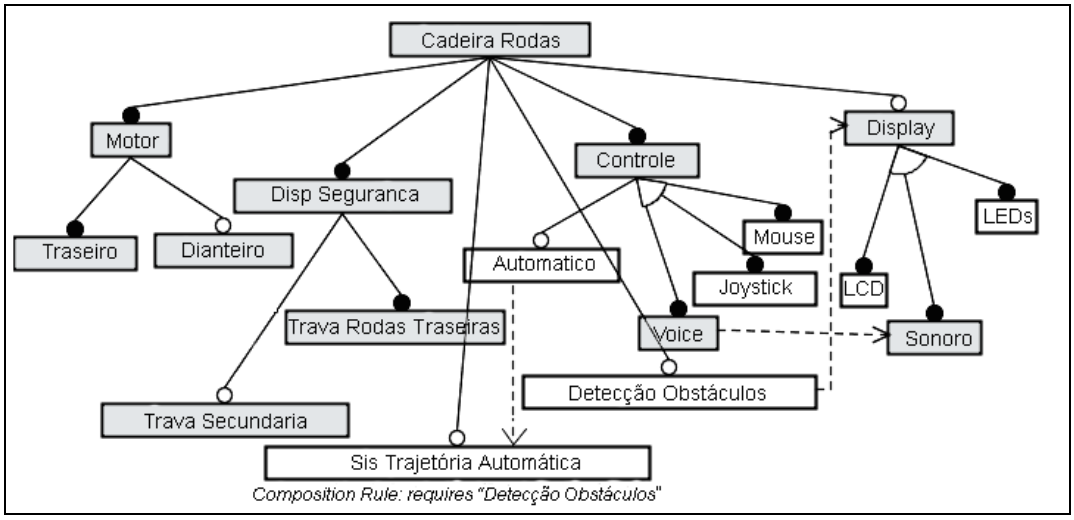


Figura 6 - Seleção de Características do FM

Como o FM não possui relação direta com a UML, há a necessidade de oferecer recursos para estabelecer esta relação. A geração de modelos para uma aplicação do domínio, usando FM e UML, somente é possível se uma relação entre estas for estabelecida. Oliveira et al. (Oliveira et al. 2004) aplicaram uma relação entre UML e FM.

Para gerar artefatos de sistemas, Oliveira et al. propuseram um Perfil UML para especificar a instanciação de frameworks orientados a objetos com base no FM e elementos UML decorados com marcações definidas por seu Perfil, denominado como UML-FI (*UML for framework instatiation*).

Para Oliveira et al.:

> "a integração do FM com a linguagem UML provou ser uma boa estratégia para descrever precisamente os mecanismos usados para associar as características de um domínio de aplicação com elementos de projeto que realizam estas características" (Oliveira et al. 2004).

4 Model-Driven-Architecture

A MDA é uma abordagem para MDD, que busca sustentar nos processos de desenvolvimento de software características como portabilidade, interoperabilidade, reutilização de modelos de projeto de softwares e de código (Miller e Mukerji 2003). Para isso, a MDA utiliza uma série de padrões definidos pela OMG.

4.1 Padrões OMG Usados em MDA

Para compreender o mecanismo adotado pela MDA para mapear e transformar modelos, há a necessidade de abordar nesse trabalho os padrões e linguagens oferecidas pela OMG, que dão suporte para a MDA. Estes padrões e linguagens têm por objetivo o intercâmbio e padronização na definição de modelos e são abordados a seguir. Existem outras padronizações da OMG que podem ser usadas na MDA, no entanto esse trabalho concentra-se apenas nas definições do MOF, OCL e XMI.

4.1.1 MOF (Meta Object Facility) Versão 2.0

O padrão *"Standard"* MOF oferece um *framework* de gerenciamento de meta-dados e um conjunto de serviços para permitir o desenvolvimento de interoperabilidade para sistemas dirigidos a modelos e meta-dados (MOF 2003). Meta-dados são dados que descrevem dados.

Segundo o documento encontrado em (MOF 2003), o objetivo primário do padrão MOF é prover um *framework* para gerência de meta-dados independentes de plataforma para a MDA, em que a notação de classe da UML é utilizada para representar os meta-modelos MOF. Meta-modelos são representações gráficas de relações que podem ser estabelecidas entre elementos descritos por meta-dados, como por exemplo, em um Diagrama de Classes UML, é possível criar elementos como: classes com atributos e métodos, relacioná-las com

outras classes, definir interfaces, fazer uma classe realizar uma interface, dentre outros. Cada um destes elementos é um meta-dado representado no MOF.

4.1.2 OCL (Object Constraint Language)

A OCL é uma linguagem formal para descrever expressões em modelos UML (OCL 2003). Expressões de OCL podem ser usadas para especificar operações/ações que, quando executadas, alteram o estado do sistema. Modeladores UML podem usar OCL para especificar restrições específicas de aplicações nos seus modelos. Modeladores UML também podem usar OCL para especificar consultas (*queries*) no modelo UML, que é completamente independente de linguagem de programação (OCL 2003). OCL não é uma linguagem de programação, mas uma linguagem de especificação formal e tem por objetivo descrever objetos e suas relações (OCL 2003).

O documento OMG que descreve a OCL (OCL 2003) expõe alguns possíveis usos desta linguagem de restrições de objetos: a) como uma linguagem de consulta; b) para especificar elementos estáticos em classes e tipos no Modelo de Classes; c) para especificar constantes para estereótipos; d) para descrever pré e pós-condições em operações e métodos; e) para descrever guardas; f) para descrever conjuntos alvos para mensagens e ações; g) para especificar restrições em operações; e h) para especificar regras de derivação para atributos em qualquer expressão sobre um modelo UML. Um documento especificado em XMI, que é explicado mais adiante, pode conter definições OCL.

4.1.3 XMI (XML Metadata Interchange)

XMI é uma linguagem para intercâmbio de informações. Ela permite o compartilhamento de objetos (XMI 2003) em formato XML. XMI pode ser útil e aplicável a uma variedade de objetos (Miller e Mukerji 2003): análise (UML), software (Java, C++), componentes (EJB, IDL, Corba Component Model) e bases de dados (CWM). O XMI é usado em todos os níveis de objetos e meta-objetos em MOF (Miller e Mukerji 2003). Ela é um padrão útil para a troca de modelos entre ferramentas de modelagem UML.

Utilizando os padrões adotados pela OMG, em especial os padrões UML, MOF, XMI e OCL, a MDA auxilia o projetista na especificação dos requisitos de um sistema em diferentes visões. Cada uma destas visões fornece detalhes diferentes de alguma característica da aplicação. Para isto, a MDA oferece recursos como:

- Especificação de um sistema independentemente da plataforma que o suporta;
- Especificação de plataformas;
- A escolha de uma plataforma particular para implementar o sistema;
- Transformação da especificação de um sistema independente de plataforma para uma plataforma particular.

A MDA recomenda que modelos de sistema sejam especificados em três visões: CIM, PIM e PSM.

4.2 CIM (Computation Independent Model)

O modelo independente de computação, ou CIM, é utilizado para descrever os objetivos do sistema. Ele oferece ao projetista uma visão dos requisitos do sistema, sem mostrar os seus aspectos computacionais. No CIM, conhecido como modelo de negócio ou modelo de domínio, é especificado o que o sistema precisa e pode fazer, sem o conhecimento da computação necessária para executá-lo. Este modelo deve ser especificado para obter uma visão de como o sistema pode ser usado e, tipicamente, é independente de como ele deve ser programado (Miller e Mukerji 2003). Ainda não são especificadas, neste modelo, restrições das funcionalidades do sistema. Ou seja, este modelo oferece ao projetista uma visão das possibilidades funcionais relevantes para o sistema em questão.

O CIM é utilizado como primeiro passo na elaboração de um sistema. Nele podem ser especificadas as características obrigatórias e opcionais, existentes em um sistema de um determinado domínio de aplicações. Para Miller e Mukerji (Miller e Mukerji 2003), o modelo independente de computação (CIM) é caracterizado como sendo um mediador entre aquelas pessoas que são especialistas em relação ao domínio e seus requisitos e aquelas que são especialistas no projeto e construção dos artefatos do sistema.

O documento oficial da OMG que explica a MDA (Miller e Mukerji 2003) não informa quais são os diagramas presentes nas visões CIM, PIM e PSM. No entanto, porque o documento referencia CIM como modelo de domínio, pode-se entender que os diagramas presentes nessa visão da MDA sejam o Modelos de *Features* (Garlan 1996, Kang et al. 1998) e o Diagrama de Casos de Uso da UML, os quais buscam capturar os requisitos do sistema. Algumas abordagens definem o termo "independente de computação" como sendo modelos que não podem gerar instâncias de objetos. Assim, o FM pode conter representações de funcionalidades e o CIM pode ser composto por muitos diagramas UML que não o de Classes e Objetos, por exemplo. A OMG não especifica quais diagramas estão presentes no CIM.

4.3 PIM (Platform Independent Model)

O modelo independente de plataforma, ou PIM, representa uma visão do sistema que contém a sua computação. Ou seja, o PIM representa os requisitos na forma de suas interações computacionais, sem, entretanto, especificar detalhes da plataforma que eles utilizam. Nesse modelo, enquadram-se RFs e RNFs que podem ser detalhados. Boas práticas da disciplina de Engenharia de Software indicam que, no modelo de um sistema, não é adequado especificar características de uma plataforma, apesar disto ser necessário na codificação do sistema. Em geral, esta prática já vem sendo adotada em projetos de software, especificando as funcionalidades dos sistemas sem vinculá-las a uma plataforma.

Existe a necessidade de documentar RNFs nos modelos de sistemas, como para sistemas embarcados (Graaf et al. 2003, Zhu et al 2003, Green e Edwards 2003, Selic e Rumbaugh 1998). A especificação dos RNFs nos modelos de sistemas restringe os RFs para as características não funcionais. Como RNFs são alterados com freqüência, por motivos diversos, os modelos do sistema devem ser modificados para refletir cada nova necessidade não funcional. Desde que os RNFs não representem características de arquiteturas, plataformas de hardware e tecnológicas, a especificação dos mesmos nos modelos do sistema continua pertencendo à visão PIM.

Requisitos não funcionais podem ser especificados nos modelos do sistema utilizando o mecanismo de marcações da UML. Estas marcações podem ser feitas decorando os elementos do modelo com estereótipos, restrições e *tagged values*. As marcações tipicamente são padronizadas pela OMG, o que significa dizer que para especificar um determinado requisito não funcional em um elemento do modelo de um sistema é recomendável utilizar as marcações definidas por ela. As marcações, tipicamente, representam RNFs e podem ser encontradas de acordo com cada Perfil definido para a UML.

Um Perfil utiliza o mecanismo de extensão da UML para representar características que não estão definidas no pacote Core do MOF. Este pacote especifica os diagramas essenciais da UML. Um exemplo de Perfil é o SPT (*Schedulability, Performance and Time*), que é utilizado para representar RNFs relacionados com restrições temporais (Becker et al. 2002, Wehrmeister et al. 2005, Selic 2005). Baseado na documentação da OMG para MDA e no SPT, o PIM pode conter especificações de restrições temporais, no entanto o projeto não pode assumir aspectos específicos de uma plataforma de software.

Uma alternativa para não tornar os RFs específicos dos RNFs é utilizar dois níveis de modelos na visão PIM. O primeiro modelo pode conter apenas os RFs do sistema, sem as marcações definidas por um Perfil. O segundo modelo pode ser decorado com os RNFs relacionados a restrições temporais, o que quer dizer que passam a atender essas restrições temporais. Caso ocorra, em um sistema, uma mudança dos seus RNFs, é possível partir de um projeto do mesmo sistema no primeiro modelo e especificar os novos requisitos não funcionais num segundo modelo. Esta prática é caracterizada como uma boa maneira de administrar os modelos do sistema, reduzindo o trabalho em caso de mudança nos NFRs de uma aplicação.

A visão CIM da MDA representa os requisitos do sistema capturados e explicitados de forma que não é avaliada a implementação destes. Na visão PIM, ao contrário da anterior, estes requisitos devem ser avaliados e especificados em diagramas. Logo, na visão PIM da MDA, Diagramas de Classe, de Objetos e de Seqüência podem ser definidos. Modelos de Classes devem ser utilizados sem a especificação das características das plataformas que podem implementá-los, isto inclui levar consigo os RNFs.

Algumas abordagens visam o desenvolvimento de *frameworks* no PIM (Oliveira 2001, Oliveira et al. 2004). Tais abordagens esperam construir aplicações com base no reuso de elementos destes *frameworks*. Para as mesmas, o conceito de Engenharia de Domínio é de fundamental importância, visto que esta objetiva a construção de software flexível para reutilização.

O PIM, baseado na documentação OMG (Miller e Mukerji 2003) e em trabalhos como os de Tekinerdogan et al. (2004), Almeida et al. (2004) e Oliveria et al. (2004), possui o legado da UML para instanciação de objetos e separa os modelos que são possíveis de gerar elementos instanciáveis dos modelos de descrição do sistema no CIM e dos modelos transformados para abrigar uma plataforma específica no PSM. Esta separação de características aumenta o reuso de modelos de um projeto de sistema, visto que um modelo independente de plataforma pode ser utilizado muitas vezes para qualquer plataforma que possa implementá-lo e executá-lo.

4.4 PSM (Platform Specific Model)

O modelo específico de plataforma, ou PSM, é uma visão do sistema dos aspectos relacionados com a plataforma que deve implementá-lo. O PSM combina a especificação do modelo independente de plataforma (PIM) com detalhes específicos de uma determinada plataforma. Neste modelo, são escolhidas uma ou mais plataformas para implementar e executar uma aplicação. Esta escolha implica definir as características necessárias para refinar o modelo PIM na(s) plataforma(s) escolhida(s), especificando como os serviços desta(s) plataforma(s) cooperam para realizar as funcionalidades do sistema.

Os elementos especificados no PSM são elementos prontos para a geração de código (Basso, Oliveira e Becker 2006). Os elementos do modelo do sistema no PSM, como possuem as características de uma ou mais plataformas, podem ser refinados para elas, o que significa dizer que podem gerar código para uma aplicação contendo características arquiteturais, de serviços, tecnológicas, etc.

4.5 PDM (Platform Description Model)

O Modelo de Descrição de Plataformas (PDM) contém as características de plataformas disponíveis para o desenvolvimento de sistemas (Willink 2003). Para transformar um PIM em um PSM, é necessário identificar as plataformas que podem efetuar este refinamento. Willink define além das visões CIM, PIM e PSM da MDA, uma quarta visão, em que é necessário documentar as plataformas que são utilizadas para desenvolver sistemas. O mesmo autor denomina esta visão como PDM.

4.6 Mapeamento e Transformação de Modelos

O PSM representa os requisitos avaliados no modelo PIM em uma arquitetura específica, que foi identificada no PDM. Nele, é possível implementar o modelo PIM em uma arquitetura específica. Na Figura 7 é mostrada a existência de um PIM, de um modelo de plataforma (PDM) e de uma transformação. O PIM é mapeado para uma plataforma e transformado no PSM.

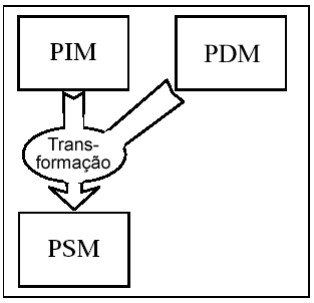


Figura 7 - Mapeamento e Transformação de Modelos

4.6.1 Mapeamento

Para efetuar a conversão de um modelo independente de plataforma para outro dependente de plataforma a MDA sugere uma série de etapas (que são detalhadas na Seção 4.6.5):

- Escolha de uma plataforma;
- Marcação dos modelos para esta plataforma e;
- Transformação do modelo para a plataforma específica.

O mapeamento pode ser realizado adicionando marcações (especificações de instâncias de modelos) nos modelos do sistema e associando os modelos marcados com algum compilador de modelos, que interpreta as marcações e gera um novo modelo dependente de uma plataforma. Um compilador de modelos pode ser uma ferramenta que recebe como entrada um modelo, realiza um processamento nos dados de entrada em busca de algumas marcações neles aplicadas e gera como saída outro modelo. Marcado o modelo, ele pode ser transformado, utilizando uma ferramenta CASE que conheça a plataforma escolhida.

Alternativamente, um compilador de modelos pode ser um elemento de um modelo que compõe a visão PDM de um sistema. Neste caso, as marcas especificadas em um elemento do modelo PDM devem possuir uma sintaxe, para possibilitar a expressão de mapeamentos e transformações de modelos entre as plataformas.

4.6.2 Mapeamento e Marcações

A Figura 7 apresenta uma única fase de mapeamento e transformação de modelos. Um mapeamento pode ser feito como exemplificado anteriormente, marcando um elemento do PIM para ser transformado para uma determinada plataforma (Figura 7). Marcações indicam como um modelo deve ser transformado.

Uma marcação pode vir de diferentes fontes (Miller e Mukerji 2003). Estas fontes podem ser: tipos de um modelo que são especificados por classes, associações ou outros elementos de modelo; regras de um modelo, por exemplo, regras de um padrão de projeto; estereótipos de um Perfil UML; elementos de um modelo MOF; elementos de modelos especificados por qualquer meta-modelo. Miller e Mukerji (2003) especificaram que marcas podem representar restrições temporais nos modelos e a fase de transformação pode escolher qual plataforma pode atender aquele requisito.

4.6.3 Mapeamentos e Templates

Um *template* pode conter muitas marcações, que especificam como um modelo deve ser transformado. Por exemplo, pode haver um *template* para o mapeamento de um modelo de sistema para *web*. O *template* deste modelo para a plataforma J2EE define as marcações para transformar este modelo em um modelo que abrigue características da plataforma J2EE. Outro *template* deste mesmo modelo, porém agora para a plataforma .NET, pode conter marcações distintas, que especificam necessidades que devem ser obedecidas na sua transformação para esta plataforma. Xdoclet (Xdoclet Tutorial 2006) é um exemplo de ferramenta e linguagem de marcação que opera com *templates*. Ela define algumas marcações que podem ser utilizadas e especializadas nos códigos do sistema, gerando o código de acordo com as marcações definidas nas linhas de um programa. É possível utilizar OCL e outras linguagens para definir *templates*.

4.6.5 As Etapas de Mapeamento e Transformação de Modelos

A cada mapeamento de um modelo, originalmente um PIM, e a transformação deste para um outro modelo, implica no mesmo tornar-se cada vez mais específico de uma ou mais plataformas. Quando são feitos mapeamentos em um modelo e transformações lhe são aplicadas, ele passa a abrigar as características especificadas no mapeamento. Isto significa dizer que o modelo vai recebendo características de implementação da arquitetura para a qual foi transformado. O último modelo, PSM, sempre é o modelo do sistema especificado para refinamento em uma arquitetura específica, ou seja, o modelo que abriga as características de uma plataforma. Um modelo pode ser mapeado e transformado muitas vezes, até que ele assuma todas as características necessárias para tornar-se um modelo válido para uma plataforma, um PSM. As etapas de mapeamentos e transformações de modelos podem ser visualizadas na Figura 8.

Almeida et al. (2004) introduziram o conceito de plataformas abstratas, para indicar modelos de nível intermediário entre um PIM e um PSM. Para representar estes modelos intermediários, foi utilizada, na Figura 8, a denominação “Modelo Intermed”. Os mesmos

recebem características não funcionais, como as de arquiteturas de desenvolvimento, mais detalhadas do que as contidas no PIM.

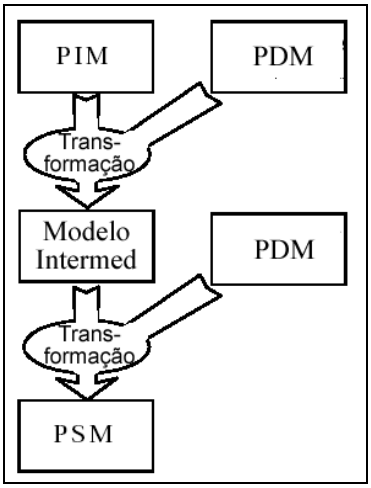


Figura 8 - Etapas de Mapeamento e Transformação

Um mapeamento é a especificação de um PIM em determinadas características de plataformas. O PIM pode ser transformado em outro modelo, mais concreto que o PIM, que especifica um determinado sub-conjunto de uma plataforma específica. Assim, o PIM é transformado para um modelo de nível intermediário e este último é transformado para um PSM (Figura 8) (Miller e Mukerji 2003, Tekinerdogan et al 2004, Willink 2004).

As relações entre as visões PIM e PSM da MDA são realizadas em etapas de mapeamento e de transformação de modelos. Estas etapas são o cerne da MDA, sendo possível estabelecer relações entre os modelos e regras para originar um novo modelo, que possa gerar código específico para implementação em uma determinada plataforma. No entanto, a MDA não determina as etapas de mapeamento e transformação. Estas etapas ainda estão em fase de pesquisa e estabelecimento de padrões pela OMG (Miller e Mukerji 2003).

A MDA também sugere que os modelos de sistemas sejam transformados de uma visão para outra. Por exemplo, de uma visão PIM o modelo de um sistema pode ser transformado para um outro modelo na visão PSM. Czarnecki e Simon (2004) classificam abordagens de MDA para transformação e divide-as em duas categorias: Model-to-Code (M2C) e Model-to-Model (M2M). Em transformações M2C o objetivo é gerar código

diretamente a partir de modelos. Em transformações M2M, um objetivo é transformar um modelo de entrada (PIM) em um outro modelo (PSM).

Na MDA, as entradas e saídas de uma transformação definem um mapeamento. Por exemplo, um PIM pode ser um modelo em que seus elementos estão decorados com marcações de um Perfil UML. Este modelo é usado como uma entrada de uma transformação, que deve gerar como saída um modelo alvo ou código para uma aplicação. Então um mapeamento descreve um modelo fonte e um modelo alvo, enquanto a transformação refina o modelo fonte no modelo alvo.

4.6.6 Categorização de Transformações

Czarnecki e Simon (2003) categorizam abordagens para transformação em dois grandes grupos: model-to-code (M2C) e model-to-model (M2M). Segundo os mesmos autores, "muitas ferramentas oferecem ambas transformações model-to-model e model-to-code (exemplos são Jamda, XDE, e OptimalJ)" (Czarnecki e Simon 2003). Em transformações M2C, o objetivo é gerar código diretamente de modelos ou *templates*. Em transformações deste tipo, o modelo do sistema pode representar as características da plataforma alvo. Por exemplo, o modelo do sistema contém características da linguagem de programação Java.

Abordagens para transformações da categoria M2C informam que os modelos prontos para a geração de código (PSMs) são transformados em código por um compilador de modelos. Um compilador de modelos processa os elementos do modelo de entrada e gera uma saída. O processamento busca elementos do modelo fonte e efetua nos elementos encontrados algumas transformações. Estas transformações podem modificar um elemento, criar novos elementos e associá-los com outros elementos.

A saída de um compilador de modelos é composta por: a) um ou mais arquivos, que podem ser representados em qualquer formato, incluindo os utilizados para intercâmbio de modelos como XMI; b) objetos que são instancias do MOF, que estão na memória do computador e podem ser persistidos de alguma maneira.

Para as abordagens de transformações da categoria Model-to-Model (M2M) o objetivo é gerar um novo modelo. O modelo gerado por transformações M2M pode ser mais detalhado que o modelo anterior (recebido como entrada pela transformação), ou até mesmo ser outro modelo contendo novos elementos de um sistema. Algumas abordagens para transformação de modelos da categoria M2M determinam que os elementos de um modelo, quando transformados por algum compilador de modelos, recebem marcações com estereótipos e restrições (Becker, Höltz e Pereira 2002).

Czarnecki e Simon (2003) salientam que transformações M2M são necessárias para preencher grandes lacunas de abstrações entre PIMs e PSMs. É mais fácil gerar modelos intermediários do que gerar diretamente, a partir de um PIM, um PSM (Czarnecki e Simon 2003). Além disso, para os mesmos autores, os modelos intermediários podem ser necessários para estabelecer otimização e ajustes no modelo (para propósitos de *debugging*) e oferecem uma visão do modelo do sistema mais especificamente para uma plataforma (Czarnecki e Simon 2003).

Czarnecki e Simon (2003) dividem abordagens para transformações M2M em outras cinco categorias:

- **Transformação de Manipulação Direta de Modelo (TMD):** abordagens desta categoria oferecem mecanismos para manipular os elementos de um modelo. Os elementos de um modelo, tipicamente, são representados utilizando o MOF. Para manipulá-los, as abordagens com foco em transformações desta categoria oferecem uma linguagem para a manipulação de modelos;
- **Relacional:** o objetivo é relacionar dois elementos de um ou mais modelos. Esta relação pode ser seguida de uma regra para transformação que determina a conversão de um modelo para o outro;
- **Baseadas em transformações de grafos:** por exemplo, o objetivo é converter um modelo representado em UML para um que use outra representação como SLOOP (Zhu, Matsuda e Shoji 2002);
- **Dirigidas por uma Estrutura de Transformações:** o objetivo é estruturar as transformações dos modelos;
- **Híbridas:** diferentes técnicas das categorias anteriores são combinadas.

4.7 MDA e Geração de Código a Partir de Modelos

A MDA também oferece recursos para a geração automática de código. Isto quer dizer que, ao selecionar uma arquitetura, como por exemplo J2EE, o modelo independente de plataforma PIM, ao ser convertido em um modelo específico de plataforma PSM, possibilita a geração automática de código para a arquitetura escolhida. Tekinerdogan et. al. (Tekinerdogan et. al. 2004) afirmam que o desenvolvimento de um sistema em MDA, utilizando o caminho mais curto para gerar código a partir de um modelo UML, procede de um CIM para um PIM, de um PIM para um PSM e de um PSM para código. Existem, atualmente, muitas ferramentas de desenvolvimento de projetos de software que convertem modelos UML em código e que aplicam alguns dos conceitos envolvidos na MDA (Mellor 2002), no entanto poucas utilizam o conceito de MDD.

Para gerar código, a ferramenta CASE deve conhecer a plataforma fim. Assim, o modelo do sistema, assumindo uma plataforma no PSM, pode ser transformado em código para aquela plataforma. Por exemplo, supondo que o PSM foi originado a partir de mapeamentos do PIM para uma plataforma Java, utilizando plataformas como J2EE e EJB. Somente pode ser gerado código para este modelo se a ferramenta utilizada for capaz de implementar estas plataformas, como a ferramenta Eclipse para a linguagem de programação Java, e interpretar o modelo PSM. O código gerado possivelmente não teria serventia, se fosse utilizada, para tal feito, a ferramenta Jude (Jude 2006), que gera código apenas para algumas características de J2SE (J2SE é parte integrante de J2EE). Além disso, a ferramenta usada para geração de código necessita conhecer o PSM, ou seja, o modelo necessita conter especificações válidas para a plataforma.

5 Trabalhos Relacionados

Este capítulo apresenta os trabalhos relacionados, separando-os de acordo com alguns conceitos necessários para MDD e usados na abordagem FOMDA (apresentada no Capítulo 4). Czarnecki e Simon (2003) categorizaram algumas abordagens para transformação de modelos na MDA. Tal categorização é utilizada para definir os trabalhos relacionados com o presente trabalho.

5.1 Trabalhos Relacionados com Transformações da Categoria M2M - Manipulação Direta de Modelos

A abordagem FOMDA aplica transformações da categoria TMD. UMT (QVT-Partners 2006), Jamda (Jamda 2003) e RDL+UML-FI (Oliveria et al. 2004) são abordagens relacionadas com a FOMDA porque elas também aplicam transformações desta categoria. É possível definir interpretadores de linguagens para transformação de modelos tanto na UMT quanto na Jamda. No entanto, isso é realizado modificando o código dos protótipos UMT e/ou Jamda. Na FOMDA Toolkit (um protótipo desenvolvido para a abordagem FOMDA) não há a necessidade disso, porque este protótipo já está preparado para a adição de interpretadores (que ocorre em tempo de execução).

A abordagem FOMDA traz inovações em relação a abordagens para manipulação direta de modelos. Ela oferece recursos para a transformação de modelos como: a) a capacidade de efetuar composição de transformadores dinamicamente; b) adicionar elementos de modelos diretamente para parâmetros dos transformadores. Em muitas abordagens, UMT e RDL_UMLFI (Oliveira 2001), para pesquisar um determinado elemento no modelo de um sistema é necessário efetuar uma varredura no mesmo. Na FOMDA este elemento pode ser identificado visualmente e ser adicionado em um determinado transformador. O protótipo FOMDA Toolkit auxilia o projetista de aplicações a realizar isso; c) a validação dos elementos mapeados para os transformadores (isto restringe os elementos que podem ser

recebidos por um transformador); d) a capacidade de determinar a execução de transformadores quando determinadas plataformas forem necessárias no desenvolvimento do sistema; f) a capacidade de personalizar uma transformação para uma determinada plataforma alvo. Esses recursos não são encontrados em abordagens para manipulação direta de modelos como UMT, Jamda e RDL+UMLFI.

Transformações da categoria de manipulação direta podem ser definidas como de baixo nível porque aplicam uma transformação diretamente nos elementos de um modelo. Transformações de alto nível podem: a) oferecer uma camada mais abstrata de transformações e mapeamentos do que as de baixo nível; b) compor, organizar e coordenar as transformações de baixo nível; c) possibilitar a visualização das transformações intermediárias entre um PIM e um PSM. Transformações de alto nível podem ser definidas como dirigidas por uma estrutura de transformação.

5.2 Trabalhos Relacionados com Transformações da Categoria M2M – Definição de Estrutura de Transformações

A abordagem FOMDA utiliza quatro etapas para organização de transformações. Estas quatro etapas são utilizadas para coordenar a transformação de um PIM para um PSM. O objetivo desta divisão é tornar o processo de mapeamento e transformação de modelos (definido pela MDA) possível de ser administrado em níveis altos de abstração. Do baixo nível (quarta etapa), em que é possível a manipulação direta de modelos, ao alto nível (primeira etapa), em que as transformações aplicadas nos modelos são coordenadas de acordo com características não funcionais de sistemas de um mesmo domínio, outras duas etapas oferecem: a possibilidade de compor transformações de manipulação direta de modelos e de adicionar visualmente elementos de modelos para as entradas dos transformadores (terceira etapa); a especificação, seleção e a troca das plataformas utilizadas por sistemas (segunda etapa).

Na organização das transformações de alto nível o objetivo é descrever o processo de transformação de um modelo de um PIM para um PSM. Para isso, é necessário especificar

características de plataformas (que são utilizadas com o propósito de aplicar transformações) e organizá-las na ordem com que as transformações devem ser aplicadas no modelo do sistema. Esta organização visa especificar os níveis intermediários de transformações compreendidos entre um PIM e um PSM, ou seja, documenta os mapeamentos do modelo do sistema para as características de plataformas disponíveis para implementá-lo em um PDM. A partir desta especificação, é possível visualizar o "caminho" que o modelo do sistema (inicialmente um PIM) deve "percorrer" entre as características de plataformas e ser transformado por elas. Isto deve ser realizado pelo projetista até que o modelo contenha as características necessárias em um PSM.

A organização das transformações de alto nível é especificada em um *workflow*, chamado de FOMDA *workflow*. O *workflow* organiza as características das plataformas, previamente especificadas em um PDM (transformações da segunda etapa), e documenta as entradas e saídas de cada característica.

Boas (Boas 2005) abordou a organização de plataformas de desenvolvimento de software utilizando um *workflow*. A idéia é semelhante à organização de alto nível de transformações que compreende a primeira etapa da FOMDA. As plataformas são *plugins* desenvolvidos para a IDE Eclipse (com base no *framework* GME) e cada uma delas é um compilador de modelos. A abordagem de Boas é adequada para especificar ordem nas transformações de um PIM para um PSM. No entanto, não é possível visualizar no seu *workflow* os tipos de modelos que podem ser manipulados pelos compiladores de modelos e nem os tipos de modelos que são gerados por estes compiladores. Além disso, não fica claro em sua abordagem como uma ferramenta deve interpretar o *workflow* e configurar os *plugins*.

OptimalJ (OptimalJ 2006) é uma abordagem para organização de transformações M2M. Ela oferece um *framework*, especificado na linguagem de programação Java, que possibilita especificar a ordem das transformações. OptimalJ é uma ferramenta que possibilita a definição de transformações de nível intermediário entre um CIM, um PIM e um PSM. A abordagem FOMDA formaliza a ordenação de transformações com o Diagrama de Atividades da UML, oferecendo uma solução para organizar as transformações necessárias entre um PIM e um PSM, independente da ferramenta MDA utilizada.

Outra abordagem para organização/estruturação de transformações é proposta por Almeida et al. (2004). Nesta abordagem as plataformas, ou compiladores de modelos, são especificados como pacotes UML. Para tanto, Almeida et al. (2004) utiliza o mecanismo de extensão da UML, para estender o elemento pacote da UML. Os objetivos desta extensão são: de aplicar transformações; e de especificar a ordem que elas são realizadas. Essa abordagem é interessante porque, assim como na abordagem FOMDA, é possível especificar a ordem das transformações utilizando diagramas da UML.

As abordagens encontradas em (Boas 2005, OptimalJ 2006, Almeida et al. 2004), utilizadas para organização/estruturação de transformações, são interessantes porque possibilitam organizar as transformações necessárias entre as três visões do desenvolvimento de um sistema definidas na MDA. No entanto, elas carecem de uma estrutura que defina uma configuração correta entre as transformações. Isto quer dizer que o projetista pode, utilizando tais abordagens, organizar transformações que não façam sentido no desenvolvimento de um sistema. Por exemplo, o projetista pode indicar que o modelo do sistema é primeiro transformado por um compilador de modelos para um modelo MVC, representado em J2EE, e depois outro compilador de modelos converte este modelo em um código para C#.

A abordagem FOMDA minimiza este problema. Isto porque as transformações que devem ser documentadas no *workflow* são representadas pelas características de plataformas identificadas no PDM. O PDM especifica as combinações válidas entre características de plataformas. Isto significa que o *workflow* ordena as transformações que já foram previamente validadas como um conjunto no PDM.

5.3 Trabalhos Relacionados com Identificação de Plataformas

Na MDA é importante identificar as características de plataformas disponíveis para implementar um sistema (Willink 2003, Tekinerdogan et al. 2004). O conceito de independência de plataformas é muito importante na MDA e somente pode ser usado se um conjunto das plataformas alvo for conhecido (Almeida et al. 2004). Na transformação de um PIM para um PSM é importante conhecer as características do PSM. Estas características

podem oferecer detalhes que devem ser adicionados num modelo de sistema. Para isso, existe a necessidade de uma representação formal das características de plataformas usadas em um MDD.

Um formalismo adotado pelos projetistas para documentar características de sistemas, dentre eles os de plataformas, é o Modelo de Features (Czarnecki 1998, Kang et al. 1998). O MF oferece um conjunto de relações que podem ser estabelecidas entre as características do modelo e, dessa maneira, estabelece uma sintaxe para a composição delas no desenvolvimento de um sistema. No entanto, o MF oferece apenas um conjunto de relacionamentos que podem estabelecer regras para a seleção das características do modelo. Algumas informações como por exemplo, "caso selecionar a característica A, então selecione B e C e se D for selecionada então selecione F e G", não podem ser especificadas apenas com as relações definidas pelo FM. Como identificado por Tekinerdogan et al. (Tekinerdogan et al. 2004) "Atualmente, na MDA a seleção de plataformas é implícita e não é provido suporte à semântica para guiar o engenheiro de software na escolha das plataformas corretas". Para tanto, o mesmo autor propôs a especificação de regras textuais nas características do FM para a composição destas em um desenvolvimento de um sistema.

A abordagem FOMDA utiliza o Modelo de Features para especificar características não funcionais de sistemas (como as que representam plataformas). Esta especificação ocorre em uma visão separada das demais da MDA, identificada por alguns autores como PDM. Tekinerdogan et al. (Tekinerdogan et al. 2004) salienta que o MF pode ser utilizado para estabelecer as transformações de nível intermediário entre um PIM e um PSM. No entanto, o mesmo autor não propõe uma solução de como utilizar o Modelo de Features para aplicar transformações. A abordagem FOMDA propõe uma solução de como utilizar o MF para aplicar transformações. Utilizando as características selecionadas do FM e as identificando como atividades em um *workflow*, a abordagem FOMDA organiza as transformações de nível intermediário entre um PIM e um PSM.

5.4 Demais Trabalhos Relacionados

O trabalho de Oliveira et al. (Oliveira et al. 2004) define uma solução para a transformação de um CIM em um PIM e de um PIM para PSM. Para tanto, o mesmo define a linguagem RDL, utilizada para efetuar a manipulação direta de modelos de *frameworks* orientados a objetos. Além disso, um perfil denominado UML-FI pode ser utilizado para acrescentar marcações nos elementos do modelo do *framework*. As marcações servem para determinar os elementos de um modelo que são obrigatórios e os que são opcionais. Transformadores podem ser especificados utilizando a sintaxe definida na linguagem RDL. Um algoritmo de um transformador (escrito em RDL) pode, por exemplo, transformar os elementos contidos no *framework* (CIM) em um novo modelo que contenha apenas os elementos de um modelo que são necessários para uma aplicação (PIM).

5.6 Análise dos Trabalhos Relacionados

Uma análise das abordagens para MDA foi realizada para identificar se elas são suficientes para aplicar um desenvolvimento dirigido por modelos. Como resultado, percebeu-se que as mesmas aplicam parcialmente o conceito de MDA e, portanto, não são suficientes para aplicar o desenvolvimento de software por meio de modelos. Este resultado levou à identificação de pontos interessantes para MDA que não são tratados nestas abordagens. Estes pontos serviram como base para a elaboração da abordagem FOMDA e são listados a seguir.

- As abordagens são focadas em transformações M2C e não em M2M. Isto impossibilita a troca de plataformas/arquiteturas sem muito esforço. Num desenvolvimento dirigido por modelos a troca de plataformas é um requisito essencial. Se as transformações são realizadas diretamente de um modelo especificado em alto nível para código, a troca de plataforma implica em perda significante do código, devendo-se partir do modelo de alto nível para realizar novamente transformações para código. Transformações M2M são mais flexíveis à mudanças de plataformas por possibilitarem que o modelo seja especificado em níveis de detalhes de plataformas;

- As abordagens não oferecem estrutura suficiente para coordenar e validar transformações M2C. Transformações M2C precisam gerar código similar ao que é gerado pelo programador. Portanto, estas transformações precisam ser coordenadas e validadas de acordo com as arquiteturas utilizadas pelo programador.
- As abordagens não oferecem estrutura suficiente para o projetista organize as transformações que ele precisa realizar no seu modelo. Em um desenvolvimento dirigido por modelos com base em MDA é necessário ter uma visão do sistema sendo transformado em baixo e alto nível. A visão de transformações de baixo nível possibilita analisar e transformar o modelo de um sistema como suas partes (Ex. adicionar um atributo em uma classe). A visão de transformações de alto nível documenta por que, para o que, onde e quais partes dos modelos devem ser transformadas (Ex. definir elementos para mapeamento objeto relacional para a plataforma Java). Esta última visão possibilita descobrir os elementos que devem ser gerados no modelo do sistema, enquanto a primeira é dedicada a descobrir como gerá-los.
- Não existe uma abordagem completa para MDA. Para aplicar um MDD é necessário integrar as soluções das abordagens estudas e também os pontos discutidos nesta Seção.

6 Abordagem FOMDA

No MDD, um modelo abstrato precisa ser refinado para um modelo concreto (Bettin 2004, Greenfield e Short 2003). Modelos refinados contêm características das plataformas que são usadas para desenvolver uma aplicação (Basso, Cavalcante e Becker 2006). Não existe na literatura pesquisada nenhuma abordagem que auxilia os projetistas no refinamento de modelos abstratos em modelos concretos para múltiplas plataformas.

Dado que o FM é usado para especificar características de plataformas e a MDA é usada para auxiliar em transformações, o objetivo desse trabalho é prover uma solução que utilize essas abordagens para compor e organizar transformações baseadas em características e, conseqüentemente, facilitar o mapeamento de um modelo mais geral para múltiplas plataformas. Assim, é proposta a abordagem FOMDA.

O principal objetivo da abordagem FOMDA é definir uma solução para projetistas identificarem mapeamentos e transformações de modelos de sistemas. Nesta abordagem, o Modelo de Features é utilizado para identificar plataformas. Estas são usadas para identificar modelos fonte (especificados em alto nível) e modelos alvo (dependentes de uma plataforma). Esta identificação representa um mapeamento. Um mapeamento determina uma transformação que deve ser realizada por uma característica do FM para, a partir do modelo fonte, gerar o modelo alvo.

Os mapeamentos e transformações de modelos são organizados em um *workflow*. Este *workflow* é uma extensão de alguns elementos do diagrama de atividades da UML. A abordagem FOMDA define um Perfil UML para organização de mapeamentos e transformações de modelos neste *workflow*.

A FOMDA Toolkit é uma ferramenta protótipo que dá suporte à abordagem FOMDA (ver Figura 9). Ela é utilizada por um projetista de uma aplicação para transformar um modelo especificado em alto nível (PIM), em um modelo dependente de plataforma (PSM) ou código para uma aplicação. O projetista de uma aplicação é um usuário da FOMDA Toolkit que pretende utilizá-la para gerar código para uma aplicação. A entrada para a FOMDA Toolkit é

um documento no formato XMI que especifica o modelo de um sistema. Este documento contém meta-dados UML representados em formato XML. Este protótipo utiliza um *parser* de XMI para converter os meta-dados, que estão em formato XML, para elementos que são instâncias do MOF.

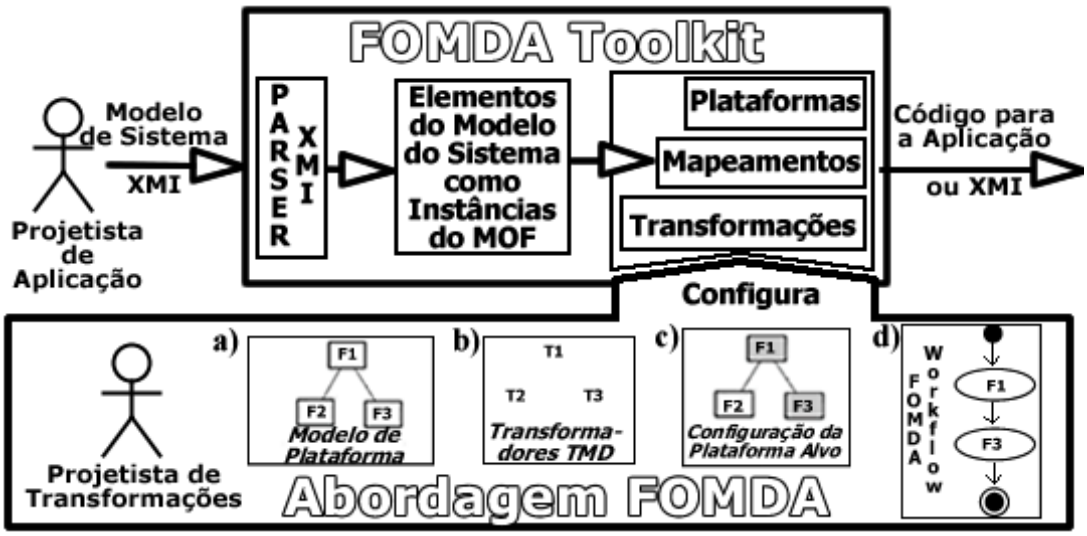


Figura 9 - Abordagem FOMDA e FOMDA Toolkit

Os meta-dados são disponibilizados em plataformas pela FOMDA Toolkit. As plataformas são características de um FM (Figura 9 (a)). Características de um FM descrevem requisitos de sistemas e podem ditar como uma transformação acontece. Estas características podem conter transformadores de modelos da categoria TMD (Figura 9 (b)), discutida na Seção 7.1. A instanciação do FM, descrita na Seção 3.3.4, provê características necessárias para desenvolver um sistema (Figura 9(c)). Estas características estão sujeitas ao uso de transformações e são organizadas em um *workflow* (Figura 9 (d)).

Para transformar um modelo especificado em alto nível, o projetista de aplicação precisa utilizar as plataformas mais os mapeamentos e as transformações definidas na FOMDA Toolkit. Esta definição é realizada por um projetista de transformações, que configura esse protótipo com base na abordagem FOMDA. O projetista de transformações é um usuário da abordagem FOMDA e a utiliza para especificar: o modelo de plataformas (PDM); os transformadores da categoria TMD; as instâncias do PDM e a organização das transformações.

Para auxiliar o projetista de transformações a configurar a FOMDA Toolkit, a abordagem FOMDA divide a organização das plataformas, dos mapeamentos e das transformações de modelos em quatro etapas. Nesta abordagem, as tarefas de identificar e instanciar o FM são realizadas na segunda etapa de organização (nesta etapa é possível especificar as plataformas). Os mapeamentos de modelos são identificados em um *workflow* que pertence à primeira etapa (alto nível) de organização. As transformações são realizadas por transformadores (explicados na Seção 7.1) e pertencem à quarta etapa de organização (baixo nível). A terceira etapa possibilita organizar transformadores de modo a compô-los dinamicamente. Detalhes desta abordagem são encontrados no próximo capítulo.

6.1 Visão Geral

O projetista de uma aplicação especifica os requisitos de um sistema, ou domínio de sistemas (utilizando a UML), e os persiste em um documento no formato XMI. O projetista da aplicação precisa utilizar este documento como entrada para a FOMDA Toolkit e usá-la para efetuar transformações em modelos de sistemas. As transformações podem originar novos modelos. Na FOMDA Toolkit, o documento no formato XMI é convertido para instâncias do MOF (ver Figura 9). Então uma transformação na FOMDA Toolkit pode gerar novos documentos XMI, instâncias do MOF ou gerar código para uma aplicação. A abordagem FOMDA auxilia o projetista de aplicações a efetuar tais transformações.

Na abordagem FOMDA, o XMI do documento do modelo de sistema pode conter elementos que representam a visão PIM da MDA. Modelos que especificam Diagramas de Classes, Diagramas de Objetos, Diagramas de Colaboração ou Diagramas de Seqüência são modelos que estão na visão PIM da MDA.

A Figura 10 apresenta como o desenvolvimento dirigido por modelos foi aplicado na abordagem FOMDA. Inicialmente, o modelo do sistema do projetista de uma aplicação está em formato XMI. Este documento é convertido pela FOMDA Toolkit para instâncias do MOF representando a visão PIM da MDA, ou seja, elementos que descrevem requisitos para uma aplicação. Transformações M2M são utilizadas em conjunto com um modelo que descreve as

plataformas que serão utilizadas para desenvolver a aplicação. Tais transformações geram modelos dependentes de plataforma (PSMs). Nestes últimos modelos transformações M2C são aplicadas, gerando código específico para o conjunto de plataformas identificadas no PDM.

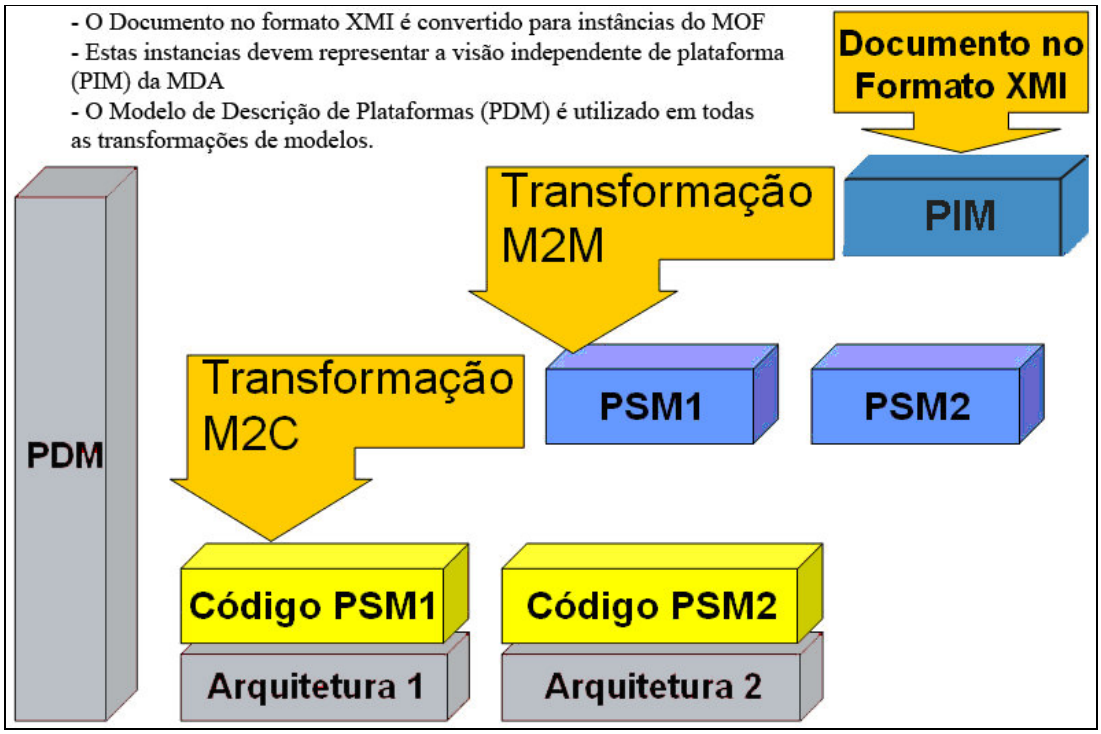


Figura 10 - O MDD Aplicado na Abordagem FOMDA

A FOMDA Toolkit foi desenvolvida para aplicar este esquema de MDD. O objetivo do projetista de aplicação ao usar a FOMDA Toolkit é converter os elementos contidos no documento XMI, que estão na visão PIM da MDA, em um PSM e usá-lo para gerar código para sua aplicação. Para efetuar a conversão de um PIM para um PSM, é necessário utilizar transformações que manipulam os elementos contidos no PIM. Para isto, é necessário conhecer os padrões da MDA, que são o MOF e XMI, descritos na Seção 4.1.

6.1.1 Transformações Independentes das Versões do XMI

O XMI é padrão de linguagem que possibilita representar elementos de um sistema, especificados com UML, em formato textual. Elementos definidos pela UML seguem o padrão MOF. Então, elementos de um modelo especificados em UML são instâncias dos

meta-dados definidos pelo MOF e são persistidos em formato XMI. Como o XMI é uma linguagem, documentos no formato XMI podem ser representados em versões diferentes da linguagem XMI. Logo, alguns elementos definidos pelo MOF são especificados de maneira diferente em cada versão da linguagem XMI. Isto ocorre porque um elemento de um modelo de sistema pode ser persistido de maneiras diferentes para cada versão de XMI utilizada.

Existe uma limitação para as transformações que manipulam elementos contidos em um documento no formato XMI. Elas funcionam corretamente quando projetadas para manipular documentos especificados em uma única versão dessa linguagem. Porém, o mesmo não pode ser garantido se estes documentos estão especificados em outras versões do XMI. Isto é um problema, porque em uma transformação M2M de manipulação direta de modelos, um algoritmo precisa ser especificado e este precisa manipular um elemento específico do modelo do sistema. Caso este algoritmo manipule um elemento do modelo especificado no formato XMI, como é realizado em abordagens como UMT, ele fica limitado a manipular os elementos de modelos que estão especificados em uma determinada versão do XMI.

Para permitir que uma mesma transformação manipule qualquer modelo de um sistema, existe a necessidade de o modelo ser representado em um formato padrão. Isto implica em evitar uma transformação manipule documentos no formato XMI. Transformações na FOMDA Toolkit podem, então, manipular elementos representados em MOF, que identifica os elementos da UML em um formato padrão. Portanto, uma solução que torna os transformadores genéricos em relação aos elementos manipulados por eles é convertê-los em objetos que são instâncias do MOF.

A abordagem FOMDA em conjunto com a FOMDA Toolkit adota tal solução para reuso de transformações da categoria M2M. O reuso é obtido porque o modelo de um sistema, especificado no formato XMI, é convertido para objetos que são instâncias do MOF. Para realizar esta conversão a abordagem conta com o auxílio da FOMDA Toolkit (apresentada com detalhes na Seção 8). As transformações na FOMDA Toolkit manipulam objetos e não elementos de um documento especificado no formato XMI. A abordagem FOMDA é utilizada para configurar as transformações na FOMDA Toolkit.

Existem algumas vantagens que as transformações da FOMDA apresentam em relação às transformações de outras abordagens: a) o rápido acesso de uma transformação aos

elementos do modelo de sistema, porque estes são armazenados na memória de acesso aleatório do computador; b) a independência das versões do XMI, utilizado para representar modelos; e, como conseqüência, c) a reutilização das transformações de manipulação direta de modelos.

6.2 As Transformações na FOMDA

A Figura 9 esboça como o projetista de uma aplicação interage com a FOMDA Toolkit para aplicar transformações em modelos de sistemas. Depois de converter os elementos do documento XMI para instâncias do MOF (com o auxílio da FOMDA Toolkit), o projetista do modelo do sistema deve manipular estas instâncias afim de transformá-las em outros elementos UML (transformação M2M) ou código para a aplicação (transformação M2C). Isso deve ser realizado, levando em conta as plataformas (encontradas em um PDM) que serão usadas para desenvolver a aplicação, também os mapeamentos do modelo de sistema para estas plataformas e as transformações que podem gerar o que é requerido pelo mapeamento. Para auxiliar o projetista de aplicação na manipulação do modelo de sistema, a abordagem FOMDA possibilita que plataformas, mapeamentos e transformações sejam previamente definidos na FOMDA Toolkit por um projetista de transformações. Portanto o esquema para MDD apresentado na Figura 10 é totalmente configurável na FOMDA Toolkit, utilizando para isso a abordagem FOMDA.

Características não funcionais descrevem os requisitos do sistema e podem ditar como uma transformação acontece. Características podem ser mapeadas para transformações, como mostrado na Figura 9 (a) e (c). A instanciação do FM, descrita na Seção 2.3.4, provê as características sujeitas ao uso de transformações e necessárias para desenvolver um sistema, como mostrado na Figura 9 (b) e (c).

Para organizar as transformações que acrescentam as características selecionadas no FM, o projetista de transformações utiliza os modelos apresentados na Figura 9 (b), (c) e (d). No entanto, para que o projetista de transformações possa especificar a ordem em que ocorrem as transformações compreendidas entre elementos numa visão PIM e uma PSM, ele

tem que analisar as seguintes questões: (i) qual é o ponto inicial?; (ii) qual é o resultado esperado?; (iii) qual é a próxima transformação? A abordagem FOMDA objetiva auxiliar os projetistas de transformações a realizarem um plano e projetarem transformações que são utilizadas por projetistas de aplicações. O plano e o projeto de transformações implicam na resposta das questões acima.

O plano envolve identificar os passos necessários para transformar um modelo de alto nível (PIM) em código fonte (Figura 9 (c)). Gerar código a partir de modelos de alto nível não é uma solução adequada quando o conjunto de características não funcionais utilizadas por um sistema for extenso (Czarnecki e Simon 2004). Nestes casos, os projetistas precisam considerar primeiro trabalhar com transformações M2M e gerar modelos intermediários entre um PIM e um PSM. Além disso, o plano para transformação de modelos é necessário para documentar todas as camadas de transformação. Deve ser informando o que é realizado em alto nível, para ter uma noção clara do que necessita ser feito antes do término de uma transformação, juntamente com a saída de baixo nível da mesma.

Para entender como os modelos identificados na Figura 9 são utilizados, é necessário saber a função que cada um exerce num desenvolvimento dirigido a modelos para a abordagem FOMDA. A próxima seção apresenta um exemplo de uso desta abordagem.

6.3 Um Exemplo de Utilização da FOMDA

Um exemplo de utilização da abordagem FOMDA pode ser visto na Figura 11. Esta última apresenta alguns modelos, que são representados como retângulos na figura e especificam uma visão dos elementos de um sistema. Estes elementos são requisitos de um sistema especificados em UML. Os modelos representam requisitos de sistemas, visões da MDA (como PIM, PSM e PDM), e modelos que definem linguagens para manipulação de modelos. Além disso, a Figura 11 contém também setas que simbolizam uma transformação dos modelos (retângulos) para outros modelos.

Para que o projetista de aplicação possa aplicar transformações no modelo de sistema é necessário que o projetista de transformações disponibilize transformadores na FOMDA Toolkit (ver Figura 9). Ele faz isso contando com as quatro etapas de organização e transformação de modelos definidos pela abordagem FOMDA.

Um transformador de manipulação direta de modelos é um algoritmo que contém código para manipulação de modelos e possibilita organizar transformações de baixo nível. Um algoritmo, tipicamente, efetua varreduras em um modelo do sistema em busca de um determinado elemento. Quando o algoritmo encontra o elemento pesquisado, ele aplica uma transformação do tipo M2C ou M2M no modelo do sistema. Na abordagem FOMDA, esse algoritmo é denominado transformador e é especificado pelo projetista de transformações.

Para escrever o código dos transformadores, a MDA recomenda o uso de linguagens de transformação de modelos. Estas linguagens são identificadas na Figura 10 pelo modelo denominado “Linguagens p/ Transformação de Modelos”. Para utilizar estas linguagens na abordagem FOMDA, é necessário o desenvolvimento de interpretadores/*parsers*. Estes últimos devem converter uma sentença especificada em uma linguagem para uma sentença utilizada pela FOMDA Toolkit. Na Seção 8.1 é descrito como é possível escrever transformadores e interpretadores de linguagens para a FOMDA Toolkit.

Uma linguagem para transformação de modelos possibilita gerar transformadores. Na Figura 11, isso é representado como uma seta que sai de “Linguagens p/ Transformação de Modelos” e chega até o modelo “Transformações M2M”. Uma transformação M2M é realizada por algoritmos de transformação.

O PDM, descrito na Seção 4.5, é um Modelo de Features que documenta as características não funcionais de um domínio de aplicações e as regras para composição entre elas. O PDM oferece, portanto, uma visão das características de plataformas disponíveis para programar um sistema e é usado na FOMDA Toolkit para documentar as plataformas. O modelo identificado como “Instância do PDM” é um Modelo de Features que contém as características não funcionais selecionadas pelo projetista de uma aplicação.

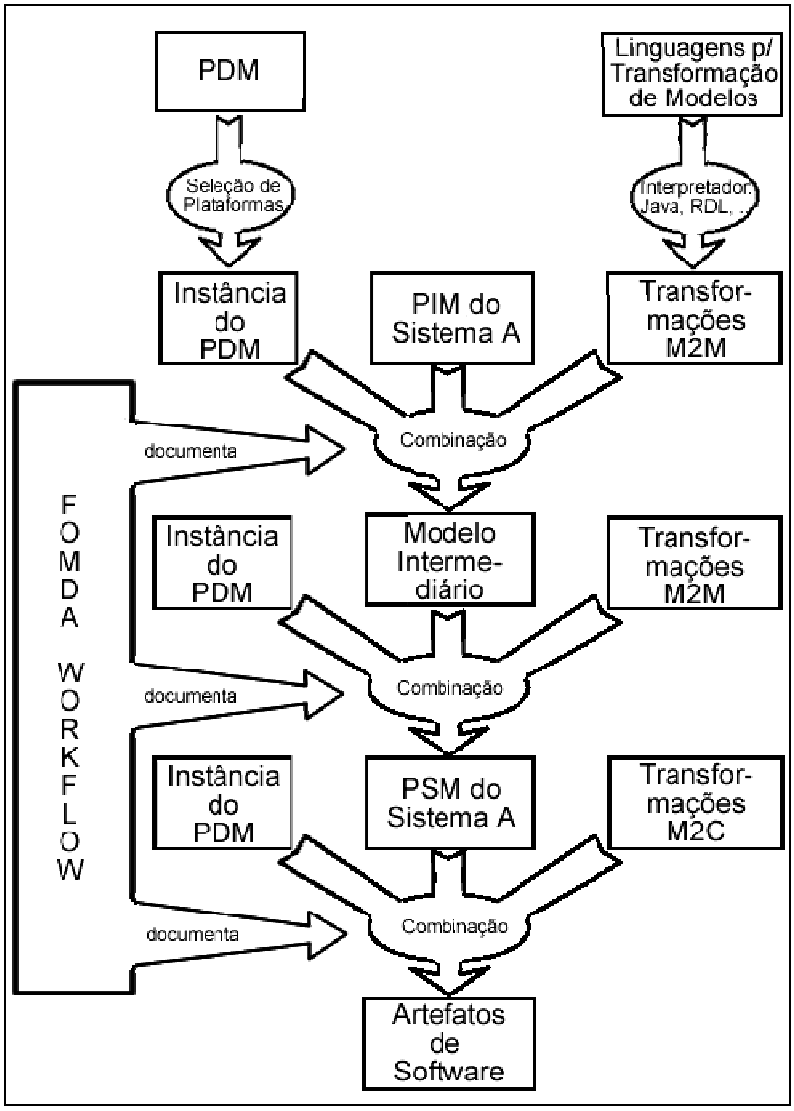


Figura 11 - Modelos e Transformações Utilizadas pela Abordagem FOMDA

O projetista de aplicação utiliza as configurações (PDM + transformadores localizados em "Transformações M2M"), definidas pelo projetista de transformações, para aplicar uma transformação em um modelo do sistema (PIM). Para tanto, o projetista de aplicação precisa: 1) selecionar as características não funcionais do MF, identificadas no PDM, gerando assim uma instância do PDM; 2) indicar qual é o PIM que representa os requisitos funcionais do sistema; 3) indicar os transformadores que ele vai utilizar para converter o PIM em um novo modelo. Estas três tarefas do projetista de aplicação são identificadas na Figura 11 pela seta que combina a saída dos três modelos (a instância do PDM, o PIM e uma transformação M2M).

As configurações do projetista de transformações definem uma receita que organiza e documenta o processo de transformações de modelos que o projetista de aplicações deve seguir. Essa ordem precisa organizar as características do FM (instâncias do PDM), os elementos do PIM e as transformações. Nesse ponto, o projetista de aplicações conta com o auxílio de um *workflow,* que documenta esta combinação.

Para organizar os modelos intermediários entre um PIM e um PSM, o projetista de transformações especifica e configura um *workflow*. Este último é denominado como FOMDA *workflow*. Ele é utilizado para fazer a combinação entre as características selecionadas no PDM, os elementos do PIM e as transformações. Cada combinação de modelos (também chamada de sincronização de modelos) e de transformações precisa ser documentada no *workflow*. O final do *workflow* é identificado pela geração dos artefatos do sistema.

O projetista de transformações pode especificar as entradas e saídas de cada transformação no *workflow*. Ele pode indicar os estereótipos, *tagged values* e restrições que os modelos transformados por uma combinação (instancia do PDM, modelo do sistema (na visão PIM) e transformação M2M) recebem após uma transformação. Além disso, é possível especificar ainda quais são as restrições que os elementos de um modelo precisam conter para serem manipulados pelas transformações. Assim, o *workflow* oferece ao projetista de aplicação uma receita para transformar PIMs em PSMs e pode documentar exatamente o que ele deve fazer em um MDD.

Os modelos da Figura 10 oferecem ao projetista de transformações a possibilidade de organizar a FOMDA Toolkit. A organização das plataformas (PDM), organização de transformações e o mapeamento de transformações para arquiteturas definem ao projetista de uma aplicação o que deve ser feito por ele na FOMDA Toolkit para transformar um PIM em código para a aplicação. Então os mapeamentos de modelos para transformações devem ser realizados pelo projetista de aplicação, seguindo a configuração definida pelo projetista de transformações.

As quatro etapas definidas na abordagem FOMDA são brevemente descritas na Figura 12. A abordagem FOMDA não trata a geração de um PIM a partir de um CIM, dado que já existem trabalhos muito bons que suprem essa necessidade (Cavalcante 2004). Logo, essa

abordagem busca solucionar a transformação de um PIM para PSMs e também de PSMs para código. Para tanto, as quatro etapas são direcionadas para o projetista de transformação que precisa configurar os mapeamentos e transformações de modelos identificados na Figura 11. Essas etapas são apresentadas na próxima Seção.

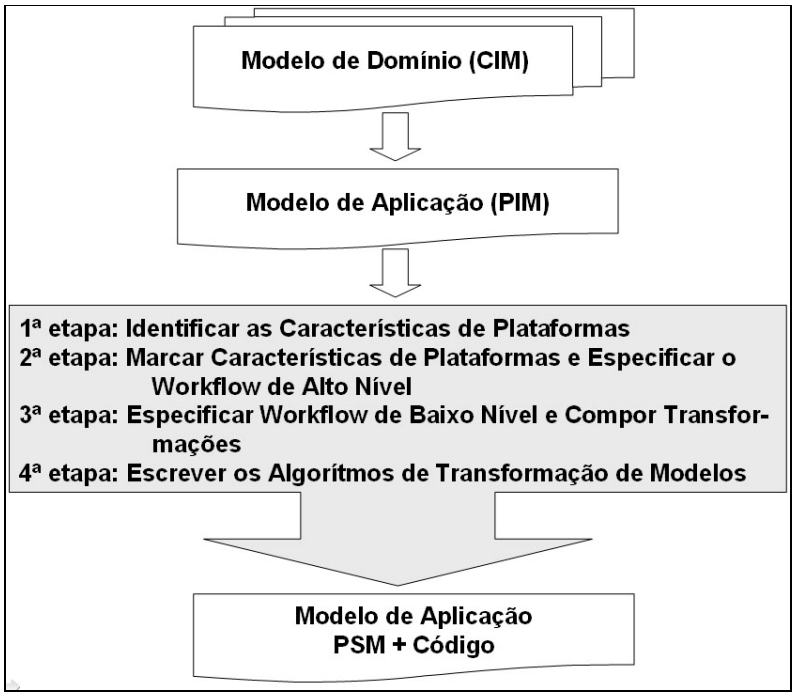


Figura 12 - Etapas Definidas na FOMDA para MDA

7 Organização das Transformações com a FOMDA

A abordagem FOMDA separa a especificação da organização das transformações em quatro etapas. A Figura 13 apresenta as quatro etapas definidas na abordagem FOMDA, identificando recursos visuais que são disponibilizados na FOMDA. Essa figura apresenta as configurações necessárias para aplicar transformações em um MDD que compõem a terceira e a quarta etapas. A segunda etapa compartilha o Modelo de Features com a primeira etapa de transformação de modelos, além de disponibilizar um *workflow* para documentar o processo de mapeamento e transformação de modelos de acordo com as características selecionadas no FM. As etapas primeira e segunda são caracterizadas como transformações de alto nível, por não efetuarem manipulações diretas em modelos de sistemas, enquanto que a terceira e quarta etapas são caracterizadas como de baixo nível, por possibilitarem a manipulação direta dos elementos do modelo de um sistema.

A primeira etapa de organização de transformações é utilizada para representar as transformações em alto nível. Neste nível de transformação, o projetista configura as plataformas alvo. Isto é realizado da seguinte maneira: as características do FM, usadas para transformação, são especificadas; em seguida, a ordem com que as características do FM efetuam transformações precisa ser descrita; por fim, os mapeamentos de um modelo fonte[2] para uma característica do FM são documentados. Isto implica em definir atividades, em um Diagrama de Atividades, para representar transformações M2M de acordo com a instância do FM.

A segunda etapa de configuração de transformações é relacionada diretamente com uma característica. Esta precisa ser configurada pelo projetista de transformações. As características do FM (definidas no PDM), que são utilizadas em transformação, são análogas às plataformas alvo na MDA (Miller e Mukerji 2003), considerando que somente características não funcionais são definidas neste modelo. Portanto, as intenções na abordagem FOMDA são as de utilizar FMs para a definição de PDMs e de possibilitar a definição de soluções genéricas para aplicar transformações.

Para que um PDM seja usado como uma solução genérica, é preciso oferecer aos projetistas de aplicações a flexibilidade para aplicar mais do que uma transformação em cada característica. A solução para isso é, além de organizar as características do PDM com a intenção de aplicar transformações, organizar também as transformações que são realizadas individualmente pelas características. Para isso, é preciso trabalhar com transformações internas às características, que é realizada na abordagem provendo transformadores às características do MF. Transformadores podem receber elementos de um modelo fonte e então aplicar transformações M2M ou M2C. As transformações no modelo de um sistema ocorrem, realmente, nos transformadores.

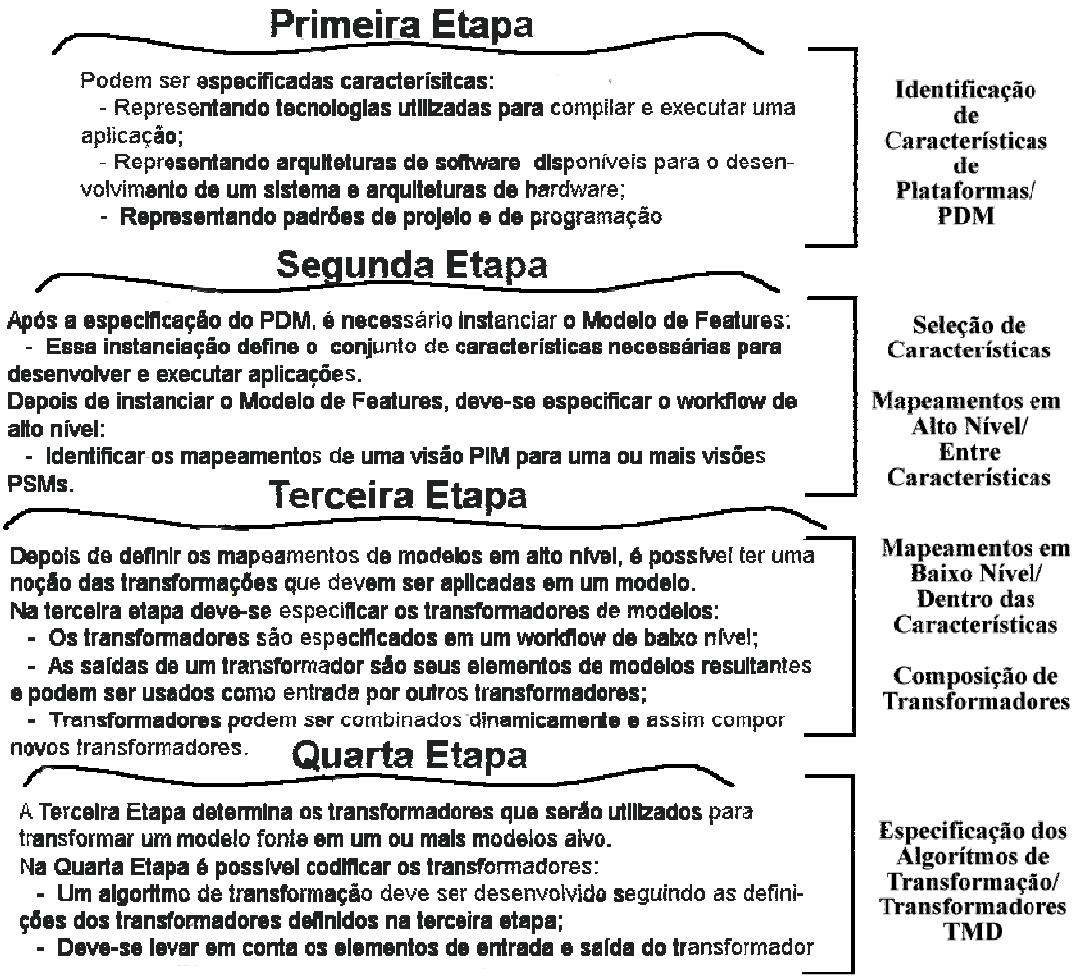


Figura 13 - Níveis de Abstração de Transformações

[2] Modelo fonte é um modelo do sistema que é recebido como entrada por um transformador.

Um transformador pode ser utilizado como interpretador de linguagens para transformação de modelos. Neste caso, ele é qualificado como um interpretador ou *parser*, porque recebe como parâmetro um código que especifica uma transformação e não um modelo fonte. Transformadores podem ser compostos para aplicar uma transformação mais específica ou acrescentar funcionalidades adicionais numa transformação. Esta é a terceira etapa de transformação, em que o projetista de transformações está preocupado em organizar as transformações de manipulação direta de modelos e também compô-las para especificar novas transformações, reutilizando as já existentes.

Finalmente, é necessário especificar o algoritmo para aplicar uma transformação. Este algoritmo é o transformador por si próprio e precisa ser escrito pelo projetista de transformações na então denominada quarta etapa de organização de transformação (o baixo nível que possibilita a manipulação direta de modelos). Este algoritmo pode ser especificado em linguagens para transformação de modelos, como RDL. No entanto, no protótipo FOMDA Toolkit, somente é possível especificar transformadores com a linguagem de programação Java, sendo necessário o desenvolvimento de interpretadores.

7.1 Quarta Etapa de Organização de Transformações

Os transformadores da FOMDA são baseados na categoria de Transformação de Manipulação Direta de modelos (TMD), definida por Czarnecki e Simon (2003). Para escrever transformadores desta categoria, o projetista de transformações precisa criar uma classe que herde de uma classe dedicada em aplicar transformações em modelos. Esta classe é disponibilizada em abordagens para MDA com foco em transformações TMD e é denominada como um transformador. Tipicamente, esta classe é abstrata e define uma operação abstrata. Esta operação deve ser sobrescrita nas classes derivadas e deve especificar o algoritmo de transformação de manipulação direta de modelos. Exemplos de ferramentas que disponibilizam classes abstratas para transformação de modelos são Jamda e UMT.

Os transformadores da FOMDA apresentam suaves modificações em relação a outras abordagens para TMD. A primeira diferença é que os transformadores da FOMDA podem

receber parâmetros, referenciados como parâmetros de transformação. Isto elimina a necessidade de efetuar uma pesquisa interna no modelo do sistema, em busca de um determinado elemento. A segunda diferença é que os transformadores podem compartilhar variáveis com outros transformadores, em tempo de execução. Por fim, a última diferença é relacionada com o FM. É possível determinar que um transformador seja executado, somente se determinada característica do FM for selecionada.

A Figura 14 apresenta parte do meta-modelo da FOMDA. Esta figura especifica os meta-dados que são usados para aplicar transformações da categoria TMD em modelos de entrada. O meta-dado denominado *Transformer* é uma interface e possui somente operações. Este meta-dado disponibiliza operações para especificar transformações que são: *execute* e *doTransformation*. Toda a classe que realiza a interface *Transformer* é um transformador na abordagem FOMDA.

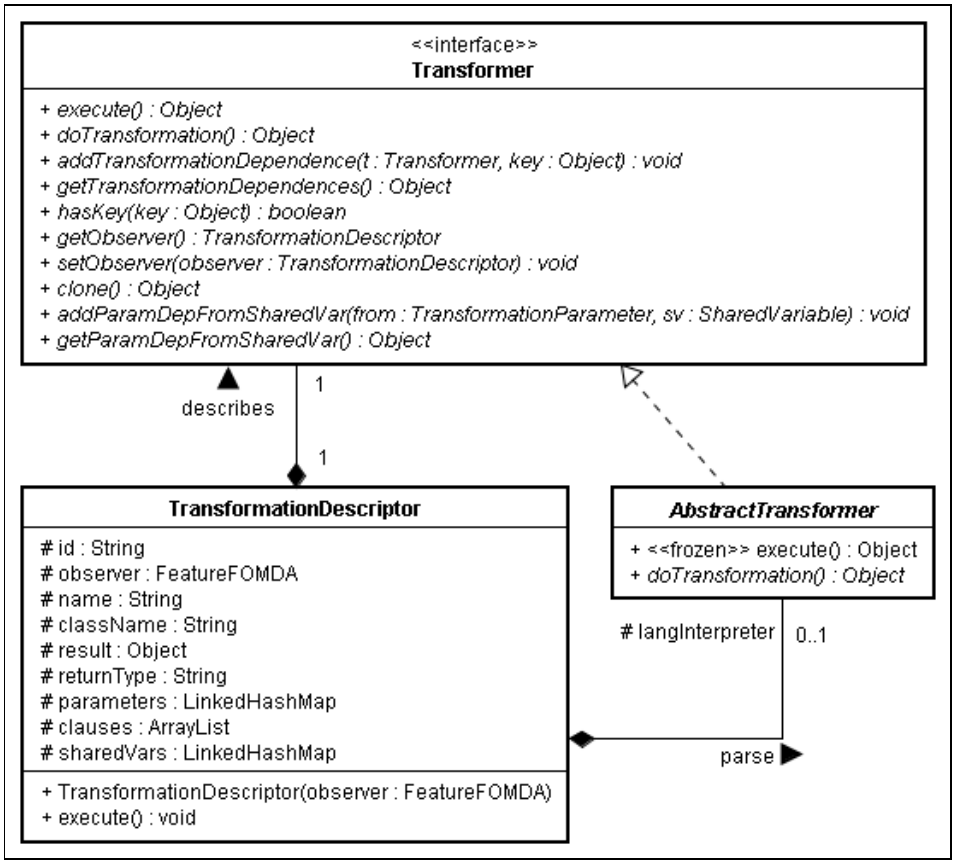


Figura 14 - Parte do Meta-Modelo FOMDA para TMD

A operação *execute* da interface *Transformer* é implementada pela classe abstrata *AbstractTransformer*. Tal operação não pode ser sobrescrita pelas classes filhas porque ela mantém a consistência na composição de transformações (assunto discutido na Seção 7.4.2). Sempre que um transformador for executado na abordagem FOMDA, a operação *execute* será invocada. Na classe *AbstractTransformer*, no final da implementação da operação *execute*, uma chamada para a operação abstrata *doTransformation* é feita. Isto implica na execução da operação *doTransformation* de uma classe derivada de *AbstractTransformer*.

A operação *doTransformation* precisa ser reimplementada em qualquer classe concreta que herde da classe abstrata *AbstractTransformer*. Os transformadores somente precisam implementar o algoritmo de transformação na operação *doTransformation*. O projetista de transformações, na abordagem FOMDA, define uma transformação em uma classe que herda da classe *AbstractTransformer* e a implementa na operação *doTransformation*.

Todo o transformador tem associado a si um descritor de transformação. Na terceira etapa de organização de transformações é possível combinar muitos transformadores, para que estes especifiquem novas transformações. Para tornar isto possível é necessário configurar cada transformador em um descritor de transformação. Na Figura 14, é possível visualizar as informações contidas em um descritor de transformação. Este é o meta-dado *TransformationDescriptor,* em que os seguintes atributos são importantes: *observer* – identifica qual é a característica do FM que contém o transformador descrito pelo descritor de transformação; *name* – o nome do transformador; *className* – identifica o nome da classe do transformador associado; *result* – armazena um objeto que é o resultado da execução do transformador associado; *returnType* – identifica o tipo do objeto resultante da execução do transformador associado.

Transformadores podem conter parâmetros de transformação. Cada parâmetro de um transformador pode armazenar um elemento do modelo do sistema. Este elemento é adicionado ao parâmetro em tempo de execução de uma transformação, pelo projetista de aplicação. Esses parâmetros podem ser utilizados no algoritmo de um transformador. Isto significa que estes podem ser utilizados em uma transformação, como, por exemplo: Um transformador ‘X’ tem um parâmetro de transformação ‘P1’. Este parâmetro pode armazenar um elemento do tipo *mof.core.Class*, ou seja, um elemento tipo classe pode ser inserido no parâmetro ‘P1’ do transformador ‘X’. Então, em tempo de execução, o projetista de aplicação

adiciona no parâmetro ‘P1’ uma classe de um modelo de sistema e executa o transformador ‘X’. No algoritmo de transformação do transformador ‘X’, o valor contido no parâmetro ‘P1’ pode ser recuperado e utilizado pelo algoritmo deste transformador.

Um exemplo de um transformador pode ser observado na Figura 15. Esta figura apresenta a classe denominada *AddClassTransformer*. Esta última é um transformador, porque herda da classe abstrata *AbstractTransformer*. Esta herança determina que a classe *AddClassTransformer* precisa implementar a operação *doTransformation* (linha 7-20). O construtor da classe *AddClassTransformer* recebe como parâmetro um objeto do tipo *TransformationDescriptor* e, dessa maneira, pode usá-lo para buscar informações a respeito da transformação.

Um objeto do tipo *TransformationDescriptor* descreve a transformação e oferece recursos úteis para o desenvolvimento do algoritmo do transformador. Estes recursos podem ser: objetos de interface gráfica, que podem ser utilizados para reportar uma mensagem ao projetista de aplicação no FOMDA Toolkit; a característica de um Modelo de Features que contém o transformador (transformadores são mapeados para características do FM); parâmetros de entrada do transformador; cláusulas que testam a seleção de características de um FM; variáveis compartilhadas do transformador, sendo possível adicionar nestas variáveis os elementos do modelo do sistema que podem ser usados por outros transformadores.

A implementação da operação *doTransformation,* mostrada na Figura 15 (linhas 7-20), contém o algoritmo de transformação do transformador *AddClassTransformer*. O algoritmo é escrito para adicionar uma nova classe em um modelo do sistema. Este modelo é recebido como parâmetro (denominado “*model*”) pelo transformador em questão. A linha 8 exemplifica como o projetista da transformação pode recuperar um parâmetro de transformação.

A classe *AbstractTransformer* contém um atributo chamado *observer*. Este atributo contém o descritor de transformação e pode ser utilizado para recuperar os parâmetros de transformação em um transformador. Isto é feito invocando a operação *getParameter* do atributo *observer*. Esta operação recebe como parâmetro um objeto do tipo *String*. Este parâmetro é o nome do parâmetro de transformação, que foi cadastrado no descritor de um transformador (isto é explicado na Seção 4.4.2). O retorno da operação *getParameter* é um

parâmetro de um transformador, que, no meta-modelo da FOMDA, é uma instância do meta-dado *TransformationParameter* (ver Figura 16).

```
1  public class AddClassTransformer extends AbstractTransformer{
2
3      public AddClassTransformer(TransformationDescriptor observer){
4         this.observer=observer;
5      }
6
7      public Object doTransformation(){
8         TransformationParameter tp = observer.getParameter("model");
9         mof.core.Model model = (mof.core.Model) tp.getElement();
10
11        mof.core.Class newClass = new mof.core.Class(model);
12        newClass.setName("TheNewClass");
13        newClass.setVisibility("public");
14        newClass.setScope("instance");
15        newClass.setAbstract(true);
16
17        model.addElement(newClass);
18
19        return newClass;
20     }
21 }
```

Figura 15 - Exemplo de Transformador TMD da FOMDA

A classe *TransformationDescriptor*, apresentada na Figura 14, é o meta-dado que representa um descritor de transformação no meta-modelo da FOMDA. Esta classe tem uma associação com a classe *AbstractTransformer*. Tal associação pode conter um objeto que é um interpretador/*parser* (se houver). Um interpretador deve ser usado quando um transformador for especificado em linguagens para transformação de modelos. Transformadores que não são instâncias do meta-dado *Transformer* precisam que um interpretador (este deve ser uma instância de *Transformer*) converta o algoritmo do transformador para instruções válidas na FOMDA Toolkit.

As transformações da categoria TMD precisam ser executadas pelo projetista de aplicação. Quando transformando um modelo de sistema os projetistas de aplicações precisam selecionar o transformador e os elementos do modelo do sistema que eles querem transformar, adicionar os elementos selecionados para os parâmetros requeridos pelo transformador e requisitar a execução do transformador. Para o projetista de aplicações, as transformações são utilizadas como foram definidas pelo projetista de transformações e, portanto, ele não precisa ter o conhecimento de como construí-las. Ainda assim, nas transformações de terceiro nível, ele pode adaptar uma transformação, efetuando uma combinação de transformadores.

A linha 9 da Figura 15 exemplifica como recuperar um elemento do modelo do sistema. Este elemento é, em tempo de execução, adicionado pelo projetista de aplicação no parâmetro de nome *"model"* do transformador *AddClassTransformer* (linha 8). A operação *getElement,* do parâmetro de transformação denominado *tp*, retorna um objeto do tipo *Object*. Este último deve ser convertido para um tipo específico e definido na linguagem java (ou, no caso do protótipo FOMDA Toolkit, num tipo de dado do MOF). A linha 9 mostra a conversão do elemento contido no parâmetro *tp* para o tipo "*mof.core.Model*".

Para especificar transformações na abordagem FOMDA, é preciso conhecer os meta-dados do MOF. Alguns dos meta-dados do MOF são definidos no protótipo FOMDA Toolkit. Neste protótipo, esses meta-dados são classes e interfaces codificadas em Java e podem ser utilizados nos algoritmos de transformação escritos nos transformadores. Os elementos do modelo do sistema são instâncias destas classes.

Da linha 11 até a 15 do algoritmo da Figura 15, uma classe é criada e adicionada no modelo do sistema. Na linha 11, um objeto (*newClass*) do tipo *"mof.core.Class"* é instanciado. Na linha 12, é especificado o nome dessa classe como "TheNewClass". Na linha 13, é especificada a visibilidade pública para o elemento criado. Na linha 14, o escopo desse elemento é definido como de instância. Por fim, na linha 15, a classe criada é definida como abstrata.

As linhas 17 e 19 (Figura 15) finalizam o algoritmo da operação *doTransformation*. A operação *addElement* é definida para qualquer elemento do MOF. Ela pode ser usada para adicionar elementos em um elemento do modelo. A linha 17 demonstra como adicionar um elemento, que foi criado no algoritmo, em um outro elemento do mesmo modelo. A linha 19 mostra o retorno do objeto denominado *model,* pela operação *doTransformation*. O valor retornado pelo transformador pode ser utilizado por outros, no que é chamado de terceira etapa de organização de transformações (apresentada na seção seguinte).

7.2 Terceira Etapa de Organização de Transformações

Realizou-se uma análise dos transformadores da categoria TMD, em abordagens como Jamda e UMT. Esta análise identificou algumas limitações nestes transformadores, como: 1) eles precisam efetuar pesquisas dentro de um modelo para encontrar um determinado elemento e aplicar uma transformação; 2) o algoritmo de transformação é estático; e 3) não são conhecidas as pré-condições para executar um transformador. É interessante conhecer o que é necessário para executar um transformador, sem recorrer ao algoritmo de transformação dele. Utilizando algoritmos de transformação estáticos, uma vez compilada uma classe que aplica transformações (um transformador), o algoritmo de transformação desta classe não pode ser modificado.

Na abordagem FOMDA, os transformadores da categoria TMD foram projetados para suprir as limitações de transformadores Jamda e UMT. Pelo momento, os recursos disponibilizados para transformadores TMD na FOMDA são parâmetros de transformação, variáveis compartilhadas, dependência de outros transformadores e cláusulas que testam se determinada característica do MF está selecionada para executar um determinado transformador. Estes recursos possibilitam a composição de transformações e podem ser utilizados na organização de transformações de terceiro nível.

Na terceira etapa de organização de transformações, Figura 13(a), o projetista de transformações tem por objetivo reutilizar transformadores TMD. Transformadores previamente desenvolvidos (como exemplificado na Seção 7.1) podem ser combinados de maneira a compor novas transformações. Para isto, um editor para composição de transformações é disponibilizado na ferramenta FOMDA Toolkit.

Uma combinação estabelecida entre dois transformadores cria um terceiro transformador. O algoritmo deste último é resultado das relações que são estabelecidas entre os dois primeiros. Um exemplo de relação que pode ser estabelecida entre dois transformadores é a dependência. Esta pode ser a dependência de um parâmetro de um transformador pelo retorno de outro. A combinação/composição de transformações é explicada nos próximos tópicos dessa seção.

Relações estabelecidas entre diferentes transformadores os agrupam em um mesmo conjunto. Este conjunto representa um novo transformador. Quando um transformador, que faz parte do conjunto de transformadores TMD, é executado, então os transformadores relacionados com ele são também automaticamente executados. Isto significa dizer que, ao compor transformações, o projetista de transformações está automatizando parte do processo de transformação de um modelo de sistema.

7.2.1 Parâmetros de Transformação

Em geral, algoritmos para transformações da categoria TMD precisam pesquisar no modelo fonte por algum elemento específico. A adição de alguns parâmetros em transformadores é usual para eliminar esta necessidade de pesquisa no modelo fonte e também para determinar os elementos que são necessários para uma transformação. Na abordagem FOMDA, um parâmetro de transformação é adicionado em um transformador através de um descritor de transformação. O descritor configura um transformador e permite a adição de parâmetros. Nestes parâmetros, elementos do modelo fonte podem ser adicionados, em tempo de execução da transformação.

Transformadores tomam como entrada alguns elementos do modelo fonte. Estes elementos podem estar contidos nos parâmetros de transformação. Um transformador pode processar estes elementos e retornar um resultado.

Parâmetros de transformação configuram pré-requisitos para a execução de um transformador. No entanto, estes parâmetros devem ser configurados no descritor de transformação. Por exemplo, é possível especificar que um transformador, para poder efetuar uma transformação, precisa receber como entrada um elemento do modelo do sistema que é uma instância do tipo *mof.core.Class*. Na FOMDA, um parâmetro de transformação é uma instância do meta-dado *TransformationParameter,* que pode ser visto na Figura 16.

A Figura 16 apresenta a classe abstrata *ParameterDescriptor*. Ela é usada como um meta-dado do meta-modelo da FOMDA para descrever um parâmetro. Este meta-dado contém três atributos: *name* - é do tipo String e serve para dar um nome para um parâmetro; *type* - é do tipo String e serve para determinar o tipo do elemento que pode ser armazenado

pelo parâmetro; *element* - é do tipo Object e armazena um objeto que é adicionado no parâmetro, em tempo de execução, e precisa ser do tipo informado no atributo *type*.

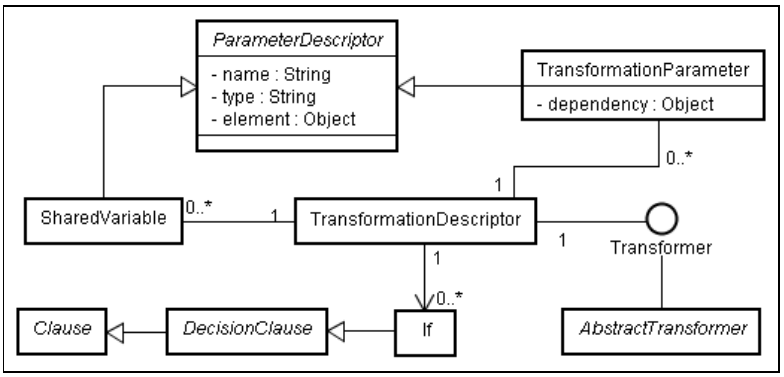


Figura 16 - Parte do Meta-Modelo FOMDA para Composição de Transformações

Utiliza-se uma restrição para determinar o tipo de elemento do modelo do sistema que pode ser adicionado no atributo *element* da classe *ParameterDescriptor*: somente elementos que são instâncias do tipo definido no atributo *type* da classe *ParameterDescriptor*. Esta restrição é definida em OCL na Figura 17.

```
context ParameterDescriptor inv:

self.element.oclAsType( self.type -> valueof() )
```

Figura 17 - Regra OCL para Meta-Dado *ParameterDescriptor*

7.2.2 Checagem de Consistência em Composição de Transformações

Para efetuar a composição de transformadores dinamicamente, como tornar possível que um transformador chame a execução de outro, é necessário avaliar a consistência entre os dois transformadores. Por exemplo, se o transformador *Y* recebe (no seu parâmetro de nome *"param1"*) o resultado do transformador *X*, então o tipo de retorno do transformador *X* tem que ser o mesmo tipo armazenado pelo atributo *type* do parâmetro de nome *"param1"* do transformador *Y*.

No meta-dado *TransformationDescriptor,* o atributo *returnType* armazena o tipo de retorno de um transformador (mostrado na Figura 14). No meta-dado *TransformationParameter,* o atributo *type* determina o tipo do objeto que pode ser armazenado no atributo *element* do parâmetro de transformação. Se no exemplo do parágrafo anterior, o retorno da transformação *X* for igual ao tipo especificado no atributo *type* do parâmetro de transformação de nome *"param1"* do transformador *Y*, então o transformador *X* pode ser adicionado no atributo *dependency* do parâmetro *"param1"*. Isto determina uma dependência válida do parâmetro do transformador *Y* pelo resultado do transformador *X*.

Essa restrição é aplicada diretamente no meta-dado *TransformationParameter* e pode ser expressa utilizando OCL. No caso da abordagem FOMDA, atualmente, o atributo *dependency* do meta-dado *TransformationParameter* pode conter somente uma dentre duas possíveis instâncias de Object (que é o tipo definido para esse atributo): ou a instância é do meta-dado *AbstractTransformer* ou do meta-dado *SharedVariable*. Logo, é preciso controlar o tipo de objeto que é instanciado para o atributo *dependency* do meta-dado *TransformationParameter*. Essa regra é mostrada na Figura 18.

```
context TransformationParameter
inv: self.dependency.oclAsType( AbstractTransformer ) and
 self.dependency.returnType -> valueof() = = self.type->valueof()
inv: self.dependency.oclAsType( SharedVariable ) and
 self.dependency.type -> valueof() = = self.type->valueof()
```

Figura 18 - Regra OCL para Meta-Dado *TransformationParameter*

7.2.3 Variáveis Compartilhadas entre Transformadores

Na abordagem FOMDA, transformadores podem compartilhar variáveis. Estas podem ser utilizadas por diferentes transformadores para aplicar transformações. Variáveis compartilhadas são instâncias do meta-dado *SharedVariable*. Esta é uma classe que herda da classe *ParameterDescriptor* e, portanto, contém os mesmos atributos que um parâmetro de transformação (com exceção do atributo *dependency*). Assim, da mesma forma que os parâmetros de transformação, variáveis compartilhadas por um transformador são configuradas no descritor de transformação deste.

É usual para um transformador delegar alguma transformação particular para outro transformador. Para permitir que isto ocorra em um transformador da categoria TMD, o mesmo precisa compartilhar uma variável com outros transformadores e chamar a execução deles (invocando-lhes a operação *execute*). Uma variável *V1*, que é usada por um transformador *X*, pode ser utilizada também por um transformador *Y* se, e somente se, o transformador *Y* tiver um parâmetro *param1,* em que o objeto armazenado no atributo *dependency* de *param1* for *V1,* e *V1* contenha um objeto no atributo *element*. Esta relação determina que *param1* depende do elemento contido em *V1* (o objeto contido no atributo *element* de uma instância do meta-dado *SharedVariable*). Além disso, para que um parâmetro de transformação dependa de uma variável compartilhada, a última das regras definidas em OCL na Figura 18 precisa ser satisfeita.

Ao contrário de um parâmetro de transformação, que recebe um elemento do modelo fonte do sistema em tempo de execução, um valor é informado para uma variável compartilhada no algoritmo de um transformador. Este algoritmo precisa recuperar a variável compartilhada para adicionar nela um elemento do modelo do sistema. Esta variável é recuperada no algoritmo de transformação pelo descritor de transformação.

7.2.4 Execução Automática de Transformações

Na abordagem FOMDA, é possível configurar alguns transformadores para que estes somente sejam executados se determinadas características do FM estão selecionadas no PDM. Neste ponto da organização de transformações de terceiro nível, o projetista de transformações ou de aplicação pode adicionar em transformadores cláusulas que testam se uma característica do PDM está selecionada. Esta é a maneira de determinar transformações que são executadas somente se determinadas características do PDM estão selecionadas. Se o teste de uma cláusula associada com um transformador for verdadeiro, então este é executado.

No meta-modelo FOMDA, foram adicionados meta-dados, que podem ser vistos na Figura 19. Estes meta-dados são usados para especificar cláusulas que relacionam características de um PDM com transformadores, testam as características e executam um transformador.

A execução automática de transformações é um ponto relevante da abordagem FOMDA, porque somente transformações relacionadas com as características previamente selecionadas no PDM são executadas. As cláusulas utilizadas para determinar quais transformações podem ser executadas são divididas em três tipos:

- Cláusulas lógicas (*And*, *Or* e *Not*);
- Cláusulas de decisão (*If* e *Else*);
- Cláusula de conclusão (*ThenClause*).

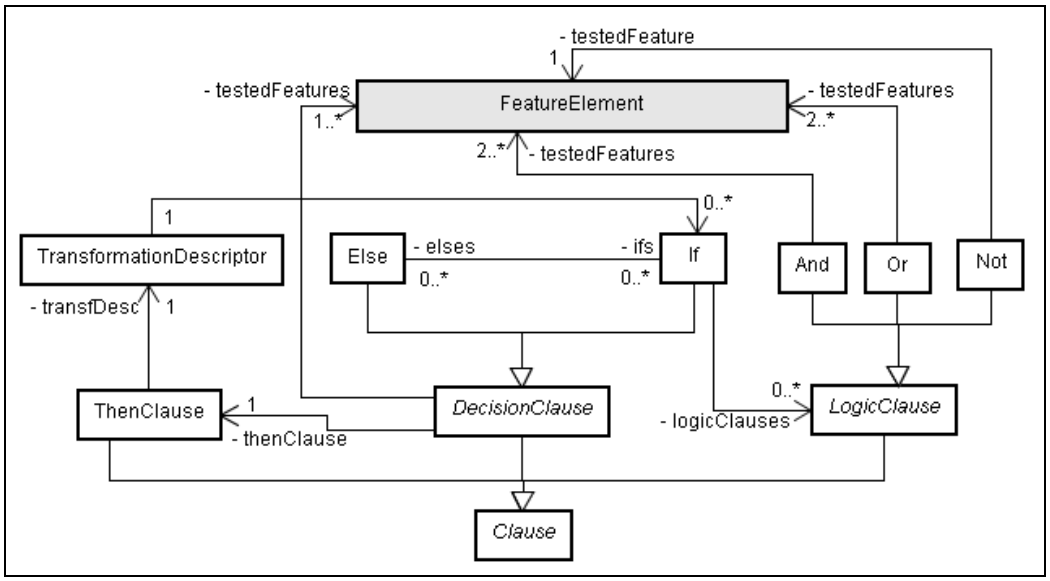


Figura 19 - Parte do Meta-Modelo FOMDA para Especificar Cláusulas

Cláusulas lógicas são usuais para testar a seleção de características de um PDM. Toda cláusula lógica precisa avaliar a seleção de uma ou mais características do PDM. Elas precisam conter instâncias do meta-dado *FeatureElement,* para permitir o teste de seleção de cada característica. Toda característica de um PDM é uma instância do meta-dado *FeatureElement*, apresentado em destaque na Figura 19. Os atributos *testedFeature* e *testedFeatures* dos meta-dados *And*, *Or* e *Not* podem armazenar este tipo de instância.

Uma cláusula lógica *"And"* (representada pelo meta-dado *And* da Figura 19) retorna verdadeiro se todas as características testadas (devem ser no mínimo duas) estão selecionadas no PDM e retorna falso se uma das características testadas não está selecionada. Uma cláusula lógica *"Or"* (representada pelo meta-dado *Or* da Figura 17) e avalia se duas ou mais características estão selecionadas no PDM. Uma cláusula "Or" retorna verdadeiro se no

mínimo uma das características relacionadas com ela estiver selecionada no PDM, do contrário ela retorna falso. Uma cláusula lógica *"Not"* (representada por uma instância do meta-dado *Not)* nega a seleção de uma característica testada, ou seja, retorna verdadeiro se a característica não está selecionada no PDM e falso se ela estiver.

Cláusulas de decisão são instâncias do meta-dado *DecisionClause*. Toda cláusula de decisão contém uma ou mais instâncias do meta-dado *FeatureElement* e uma instância do meta-dado *ThenClause*. Isto significa que as cláusulas de decisão avaliam se uma ou mais características do PDM (que são instâncias de *FeatureElement*) estão selecionadas. Caso todas as características avaliadas por uma cláusula de decisão estiverem selecionadas, então uma cláusula *"Then"* é acionada.

Um descritor de transformação pode ter associado a ele zero ou mais cláusulas *"If"*. Estas são instâncias do meta-dado *If*. Cláusulas "*If*" são cláusulas de decisão e podem conter zero ou mais cláusulas lógicas e zero ou mais cláusulas *"Else"*.

Cláusulas *"Else"* são instâncias do meta-dado *Else*. Estas cláusulas são testadas apenas se o retorno de uma cláusula *"If"* (relacionada com a cláusula *"Else"* ou com uma cláusula *"Else"* antecessora) retornar falso. Cada cláusula *"Else"*, subseqüente de uma cláusula *"If"*, somente é testada se a cláusula *"Else"* antecessora não retornar verdadeiro.

Cada cláusula de decisão precisa ter associada a ela uma cláusula *"Then"*. Cláusulas *"Then"* são instâncias do meta-dado *ThenClause* e são usadas para determinar a execução de algum transformador, identificado pela associação entre os meta-dados *ThenClause* e o *TransformationDescriptor*. Então, cláusulas *"If"* e *"Else"* podem, se suas condições de teste retornarem verdadeiro, chamar a operação *execute* de uma cláusula *"Then",* para executar o transformador associado com a mesma. Uma cláusula *"Then"* contém um descritor de transformação associado. Quando esta cláusula é acionada, isto implica na invocação da operação *execute* do descritor de transformação associado a ela.

7.2.5 Recursos Desejáveis em Transformações de Baixo Nível e Possíveis Soluções

Os recursos comentados na Seção 7.2 possibilitam a composição de transformações dinamicamente. No entanto, tais recursos são limitados e não possibilitam especificar

composições mais complexas. Neste sentido, este tópico apresenta alguns recursos que são desejáveis para a composição de transformações que não foram devidamente especificados neste trabalho. Além disso, são apresentadas possíveis soluções para adicionar esses recursos na organização de transformações de terceiro nível.

O uso de cláusulas por transformadores determina a execução de outros transformadores apenas se algumas características do PDM estão selecionadas no modelo. Podem ser utilizadas mais de uma cláusula em um mesmo transformador. Isto significa que é possível executar mais de um transformador. Muitas vezes é interessante ordenar a execução das transformações relacionadas às cláusulas. No entanto, não é possível especificar ordem para a execução de transformadores relacionados com cláusulas, na organização de transformações de terceiro nível.

Um recurso que pode estar embutido em transformadores TMD é permitir que determinados trechos de um algoritmo de um transformador sejam especializados por outros transformadores. É desejável que, em determinadas linhas de um algoritmo de transformação, seja possível delegar uma transformação para outro transformador. Isto é possível de ser feito em transformadores TMD na abordagem FOMDA, porém de maneira estática, o que significa dizer que isto é realizado no código do transformador. É interessante determinar qual é o transformador que é executado, em uma determinada linha de um algoritmo de transformação de um outro transformador, dinamicamente. No entanto, isto não é possível de ser realizado com os recursos oferecidos em organização de transformações de terceiro nível. Uma possível solução é definir pontos de especialização dentro do algoritmo de transformação de um transformador.

Pontos de especialização são linhas ou trechos de um algoritmo de transformação, que podem ser executadas por outros transformadores. Transformadores podem ser adicionados nestes pontos dinamicamente. A operação *execute* destes transformadores pode ser invocada quando o ponto de especialização for alcançado na execução do algoritmo de transformação. Para tanto, é necessário definir nos pontos de especialização, as interfaces que os transformadores que especializam cada ponto do algoritmo precisam implementar. Um estudo mais aprimorado e detalhado precisa ser realizado para adicionar estes recursos em transformações de terceiro nível.

7.3 Primeira Etapa de Organização de Transformações

Na MDA, a identificação das plataformas alvo, utilizadas para desenvolver um determinado sistema, é uma tarefa importante. As plataformas alvo identificam características que um PSM deve conter. Portanto, estas plataformas podem determinar transformações que são aplicadas em um PIM até que ele contenha detalhes que o colocam em uma visão PSM.

Willink e Boas (Willink 2003, Boas 2005) introduziram na MDA uma quarta visão (apresentada na Seção 4.5), denominada PDM, em que é possível especificar plataformas alvo de um domínio de sistemas. A abordagem FOMDA utiliza as características especificadas pelos projetistas de transformações no PDM, para auxiliar o projetista de aplicações na transformação de um PIM para um PSM. A especificação do PDM é categorizada como a primeira etapa de organização de transformação na abordagem em questão.

A visão PDM pode ser utilizada pelo projetista de transformações para especificar os requisitos disponíveis para o desenvolvimento de softwares de uma família de sistemas. Nesta visão, é possível especificar requisitos de um domínio relacionados com características de arquiteturas, serviços, tecnologias, hardware, etc. Abordagens centradas em GP, em especial as que utilizam ED e PLA, auxiliam o projetista de transformações a identificar características de um domínio de sistemas e, portanto, podem auxiliar na especificação do PDM.

Abordagens centradas em GP utilizam o Modelo de Features (FM) para identificar características de sistemas. Logo, o FM é o diagrama mais adequado para modelar plataformas alvo, utilizadas no desenvolvimento de software, e pode ser utilizado para especificar a visão PDM da MDA. Por definição, o PDM é análogo a um Modelo de Features, que contém características de plataformas. Para especificar o PDM, a abordagem FOMDA utiliza o Modelo de Features pelos seguintes motivos: ele é bem aceito pelos engenheiros de software em abordagens para PLA; possibilita que características de plataformas sejam identificadas no diagrama; e oferece sintaxe para a seleção das características do modelo. Esta seleção pode determinar a configuração das características de plataformas alvo para desenvolver uma aplicação.

As características de um PDM, que são selecionadas no processo de instanciação (explicado na Seção 3.3.4), configuram um produto. Ou seja, elas determinam aquilo que deve fazer parte de uma aplicação. Tipicamente, em uma visão PDM onde é utilizado o FM para especificação dos requisitos de sistemas, somente características não funcionais (como as que descrevem plataformas alvo) são especificadas. Então, as características que são selecionadas no PDM configuram os recursos das plataformas alvo que devem ser utilizados no desenvolvimento de um sistema. A abordagem FOMDA utiliza o FM na visão PDM, porque este permite a configuração semi-automatizada das características das plataformas alvo (desde que exista uma ferramenta que o interprete, como a FOMDA Toolkit). A instância do PDM determina a transição da primeira para a segunda na abordagem FOMDA.

Na abordagem FOMDA, características do Modelo de Features (na visão PDM) podem ser mapeadas para transformações. Qualquer característica selecionada no PDM pode aplicar uma transformação em um modelo fonte, gerando um novo modelo ou código. Portanto, uma característica é um compilador de modelos, que contém transformações. Estas últimas são identificadas na Figura 20 por T1, T2 e T3.

Uma instância do PDM (Modelo de Features com características selecionadas) determina um conjunto de características que configuram uma plataforma alvo. A configuração formada por este conjunto determina as características que devem ser utilizadas para desenvolver uma aplicação. Logo, as características selecionadas podem ser utilizadas para organizar transformações em alto nível porque documentam com exatidão as arquiteturas/plataformas às quais o modelo de um sistema deve ser transformado. Portanto, o projetista de um sistema pode se beneficiar do PDM para transformar o seu modelo de uma visão idependente de plataforma (PIM) em uma visão dependente de plataforma (PSM).

A Figura 20 apresenta um exemplo, em que as características *"Característica A", "Característica B"* e *"Característica C"* compõem uma plataforma alvo. Essa figura demonstra que é possível utilizar o PDM para auxiliar em transformações de modelos. A mesma figura apresenta uma entrada de um modelo na visão PIM para "máquina" representada por um retângulo contendo a visão PDM (um FM). Esta máquina transforma o modelo de um sistema na visão PIM para outro na visão PSM. Esta transformação é realizada pelas três características selecionadas no PDM, pois cada uma destas contém um transformador. No exemplo da Figura 20, a *"Característica A"* contém o transformador *T1*.

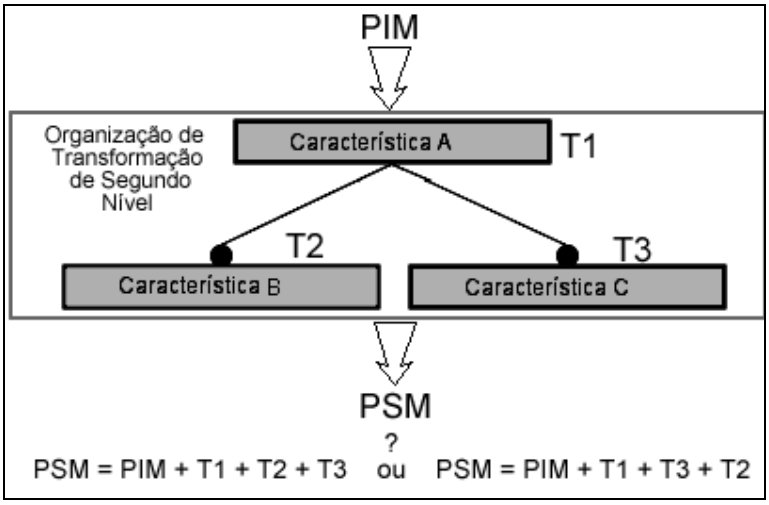


Figura 20 - Organização de Transformação de Segundo Nível

Tanto o PIM quanto o PDM podem representar requisitos de qualquer domínio de aplicações, desde que o PIM utilize algumas características definidas no PDM (que é especificado para um domínio particular). Portanto, o PDM deve conter um conjunto genérico de características, que pode ser aplicado para o desenvolvimento de qualquer aplicação pertencente ao domínio que ele representa. Este modelo recebe como entrada um PIM qualquer e o transforma em um PSM, que contém as características selecionadas no FM.

As características *"Característica A"*, *"Característica B"* e *"Característica C"* destacadas na Figura 20 são utilizadas para transformar um modelo de entrada (PIM) em um modelo refinado (PSM). Tais características estão "mapeadas" para as transformações T1, T2 e T3, então o modelo resultante das transformações (PSM) é o modelo de entrada (PIM) somado às transformações T1, T2 e T3. O mapeamento de uma transformação para uma característica do FM ocorre pela adição de transformadores à característica. O projetista de transformações desenvolve transformadores no terceiro e quarto níveis de organização de transformações da abordagem FOMDA. Estes transformadores são organizados dentro de características do FM. Portanto, o FM configura um conjunto de transformações que visam gerar modelos intermediários entre um PIM e um PSM ou o próprio PSM.

O FM pode ser utilizado para aplicar transformações no modelo fonte de um sistema. No entanto, ele não especifica a ordem em que as transformações são aplicadas no modelo fonte. Por exemplo, na Figura 20, não é possível saber qual é a ordem em que as transformações são aplicadas no PIM: se primeiro T1 depois T2 e por fim T3 ou se primeiro

T1 depois T3 e por fim T2. A ordenação das transformações é interessante, porque, por exemplo, T3 pode precisar de um modelo mais refinado que o PIM (que contenha alguns estereótipos) e T2 pode gerar o modelo mais refinado que T3 precisa receber como entrada. Neste caso, não há sentido em aplicar a transformação T3 no modelo do sistema antes de T2. Então, a solução para este caso é especificar a ordem para a geração do PSM, como, por exemplo, PSM = PIM + T1 + T2 + T3. A abordagem FOMDA utiliza o primeiro nível de transformação para documentar esta ordenação.

7.3.1 Especificação do PDM

Na abordagem FOMDA, o Modelo de Descrição de Plataformas (PDM) é um Modelo de Features. O objetivo do projetista de transformações ao especificar o PDM é identificar características não funcionais que podem ser utilizadas para desenvolver sistemas em uma organização. Características que ele pode especificar nesta visão são relacionadas com arquiteturas de desenvolvimento, padrões de projeto, serviços de software, tecnologias de desenvolvimento, hardware, restrições temporais, etc.

O objetivo do projetista de transformações, na primeira etapa de organização de transformações da FOMDA, é determinar um domínio tecnológico e arquitetural, que o projetista de aplicações pode utilizar para desenvolver qualquer aplicação deste domínio. A primeira etapa de organização de transformações é baseada na abordagem de Tekinerdogan et al. (Tekinerdogan et al 2004).

A Figura 21 apresenta um exemplo de PDM, no qual características não funcionais, utilizadas para desenvolver aplicações em uma organização qualquer, foram especificadas. Este é um exemplo simples e especifica que: (1) o uso de padrões de projeto (*design patterns*) é opcional, mas caso esteja selecionado no modelo, podem ser utilizados os padrões *MVC* e *Observer*; (2) o desenvolvimento de uma aplicação precisa da escolha de uma linguagem de programação, que deve ser *Java* ou *VB.NET*; (3) toda a aplicação precisa conter uma interface gráfica de usuário, que deve ser desenvolvida com *Swing*, *JSP* ou *ASP*.

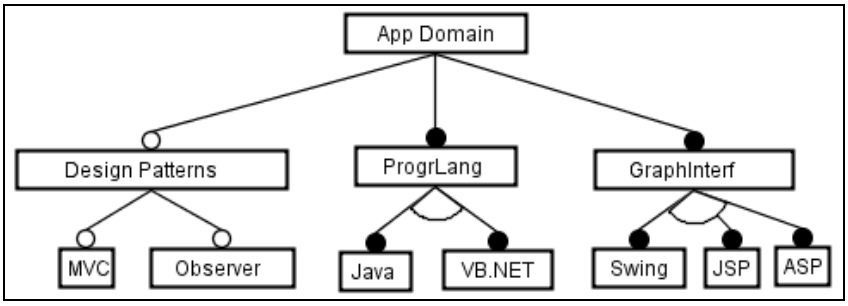


Figura 21 - Exemplo de PDM para a Abordagem FOMDA

É determinado que, se a linguagem de programação escolhida for *Java*, então a interface gráfica não pode ser *ASP*. Da mesma maneira, se a linguagem escolhida for *VB.NET*, a interface gráfica da aplicação não pode ser *Swing* nem *JSP*. Não é possível especificar estas regras graficamente no FM. Porém, como foi explicado na Seção 2.3.2, *composition rules* podem ser adicionadas nas características do FM. Isto pode ser feito nas características *Java* e *VB.NET* (como é mostrado na Tabela 1), em que as regras para seleção destas características são especificadas textualmente como *composition rules*.

Tabela 1 - Regras Para Composição de Características

Característica	Composition Rule
Java	excludes “ASP”
VB.NET	excludes “Swing” and “JSP”

7.3.2 Instanciação do PDM (Transição da Primeira Etapa para a Segunda)

O PDM é definido pelo projetista de transformações e utilizado pelos projetistas de aplicações. Com base no PDM definido na Figura 21, o projetista de aplicações está limitado a desenvolver software utilizando os recursos identificados nesta figura. Para desenvolver um sistema, o projetista de aplicações seleciona as características do PDM que quer utilizar. Esta tarefa determina a transição da primeira etapa de organização de mapeamentos e transformações e modelos para a segunda das etapas definidas na abordagem FOMDA. A Figura 22 mostra em destaque características que poderiam ter sido selecionadas por um projetista de aplicações para desenvolver sistemas.

Na abordagem FOMDA, a partir desta seleção, o PDM é utilizado para aplicar transformações em um modelo de sistema. O PDM recebe como entrada um PIM e o transforma em um PSM. Isto é realizado com o auxílio da ferramenta FOMDA Toolkit, de maneira semi-automatizada, ou seja, é necessária a intervenção do projetista de aplicação.

Cada característica selecionada no PDM precisa efetuar uma transformação. Uma transformação pode adicionar, por exemplo, marcações no modelo de entrada, que podem ser utilizadas por outras transformações para gerar o PSM. Neste sentido, é importante identificar quais são as transformações que cada característica deve realizar num modelo de sistema.

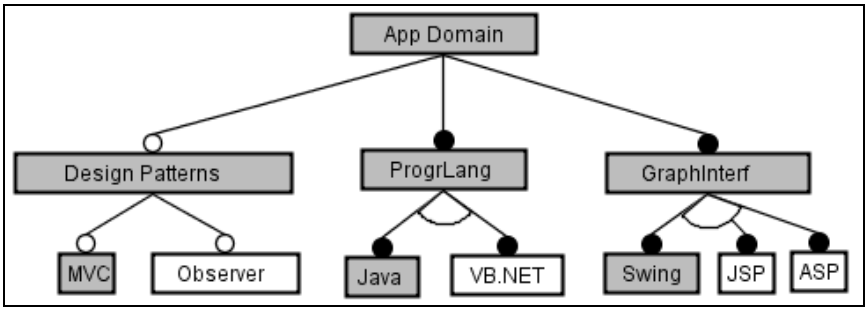


Figura 22 - Instância de PDM Usada para Desenvolver Aplicações

7.3.3 Mapeamento de Transformações em Características do FM

Transformações são transformadores especificados na terceira e quartas etapas de organização de transformação e podem ser adicionadas nas características do FM. Isto significa que mapeamentos de transformadores da categoria TMD podem ser utilizados por cada característica do modelo. Características do FM não possuem relação direta com transformações nas abordagens estudas na literatura (Czarnecki 1998, Czarnecki e Simon 2003, Deeltra et al. 2003). Logo, o meta-modelo de Features, mostrado na Figura 23, foi estendido para suportar transformadores, como é mostrado na Figura 24.

A Figura 23 apresenta o meta-modelo de Features, especificado com base no Modelo de Features de Czarnecki (Czarnecki 1998). Na Tabela 2, a coluna “Meta-dado” contém os meta-dados (definidos na Figura 23) que são utilizados para especificar os possíveis relacionamentos que podem ser estabelecidos entre as características do Modelo de Features de Czarnecki.

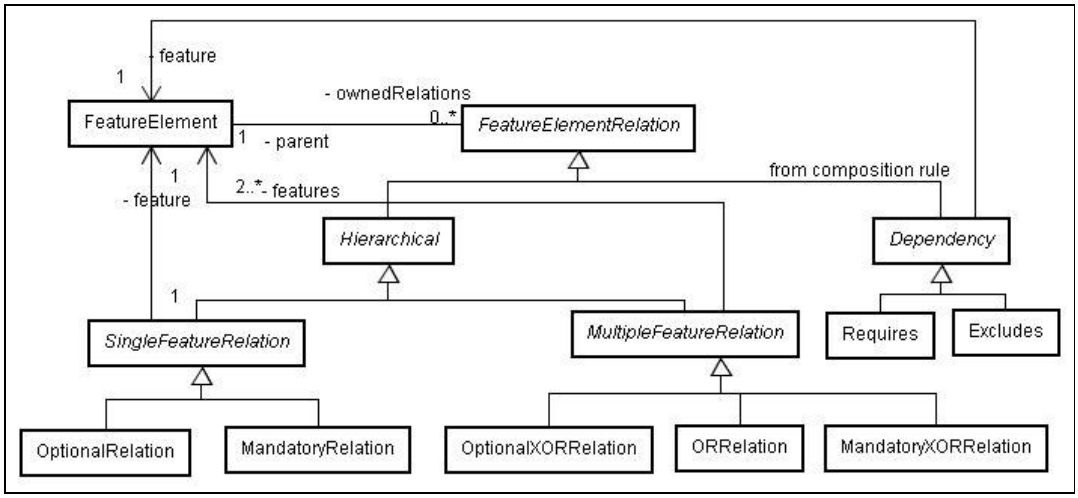


Figura 23 - Meta-Modelo de Features

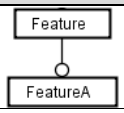
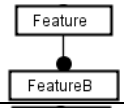
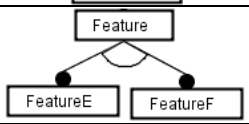
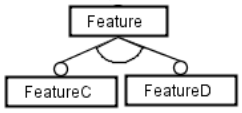
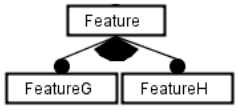
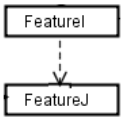
Na abordagem FOMDA, transformadores podem ser mapeados para características do FM. O meta-modelo de Features foi incorporado e estendido no meta-modelo FOMDA. A Figura 24 apresenta extensões aplicadas no elemento FeatureElement, em destaque nesta figura. O meta-dado *FeatureElement* foi estendido pelo meta-dado *FOMDAFeatureElement*.

FOMDAFeatureElement é o meta-dado que especifica uma característica do FM, com extensões para aplicar transformações. Este meta-dado possui um relacionamento com o meta-dado *TransformationDescriptor*, que descreve uma transformação. Uma instância do meta-dado *TransformationDescriptor* precisa conter uma instância do meta-dado *Transformer*, que é o transformador da categoria TMD. Uma característica de um FM pode aplicar mais de uma transformação e, para isto, ela deve disponibilizar mais de um transformador.

Analisando as características em destaque na Figura 22, é possível ter uma noção das características que são utilizadas para transformar um PIM em um PSM. Como discutido anteriormente, para transformar um PIM em um PSM, é necessário especificar ordem nas transformações. Para identificar a ordem das transformações é necessário identificar antes as entradas e saídas de cada característica do FM. Isto implica em identificar que tipo de elemento uma característica do FM pode adicionar no modelo fonte do sistema (após uma determinada transformação). Além disso, também é necessário identificar as marcações nos elementos do modelo de sistema. Estas são utilizadas por uma característica do FM para

aplicar transformações. Portanto, identificar o que uma característica recebe como entrada e o que ela gera como saída é uma tarefa importante. A abordagem FOMDA resolve isso adicionando às características parâmetros de entrada e de saída.

Tabela 2 - Relacionamentos e Respectivos Meta-dados do Modelo de Features

Nome da Relação	Descrição	Representação Gráfica	Meta-dado
Optional Relation	A característica relacionada pode ser selecionada		OptionalRelation
Mandatory Relation	A característica relacionada sempre é necessária no modelo		MandatoryRelation
Mandatory Mutually Exclusive Relation	É obrigatória a seleção de apenas uma característica dentre as demais características definidas na relação		MandatoryXORRelation
Optional Mutually Exclusive Relation	É opcional a seleção de apenas uma característica dentre as demais características definidas na relação		OptionalXORRelation
Or	No mínimo uma característica, dentre as demais definidas na relação, precisa ser selecionada		OrRelation
Requires	Se uma característica está selecionada e ela contém este tipo de relação, então a característica relacionada com ela precisa também ser selecionada		Requires
Excludes	Se uma característica está selecionada e ela contém este tipo de relação, então a característica relacionada com ela não pode ser selecionada	Não Possui Representação Gráfica	Excludes

O meta-dado *FeatureParameter* pode ser utilizado para documentar parâmetros de entrada e saída de cada característica. Ele é um tipo de *ParameterDescriptor* e, portanto, contém um nome e descreve o tipo de objeto que pode ser armazenado no atributo *element*. Além disso, um parâmetro de uma característica pode conter marcações, que são estereótipos restrições ou *tagged values*. Estas marcações servem para documentar, de maneira detalhada, os elementos de modelos que podem ser recebidos ou gerados por uma característica. Como cada característica pode conter muitos transformadores e os transformadores podem receber parâmetros distintos, as características também o podem. Além disso, é possível especificar mais de uma saída para uma característica, visto que transformadores podem efetuar transformações diferentes e, portanto, gerar modelos diferentes como resultado. A adição de

parâmetros de entrada e saída às características do FM é um ponto positivo na abordagem FOMDA, porque documenta em alto nível as entradas e saídas das transformações.

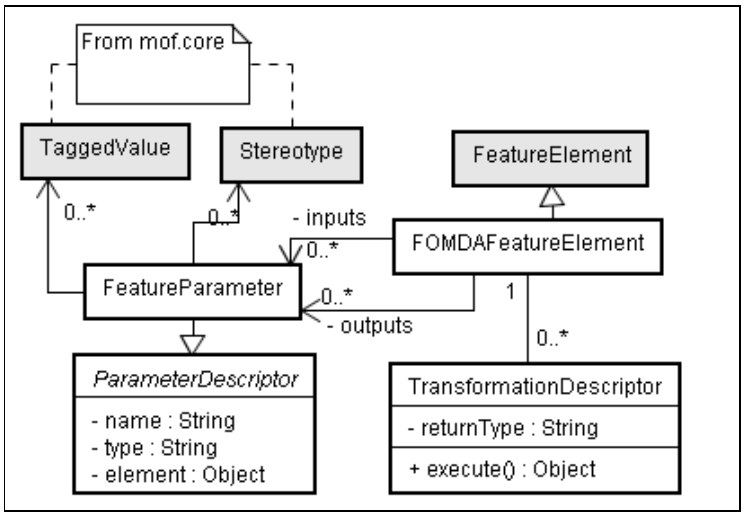


Figura 24 - Extensões do Meta-Modelo de Features

7.4 Segunda Etapa de Organização de Transformações

Transformações de nível intermediário, como consecutivas transformações PIM-para-PSM, requerem um plano de transformação. Este plano precisa documentar as transformações aplicadas entre um modelo fonte e um modelo alvo, além da ordem em que elas são executadas no contexto geral das transformações. Como a FOMDA é uma abordagem para organização de transformações, em que quatro etapas de transformações podem ser utilizadas para transformar um PIM em um PSM, o projetista de transformações pode fazer este plano de acordo com cada etapa de transformação. Quando modelando o plano de transformações em alto nível, o projetista de transformações especifica as entradas e saídas das plataformas, os mapeamentos do modelo fonte para as mesmas e a ordem com que estas últimas são usadas para aplicar transformações no modelo fonte. Quando projetando o plano de transformações de baixo nível, é mais relevante o mapeamento de elementos do modelo fonte para os parâmetros de transformação, transformadores, etc.

Para desenvolver sistemas que são dirigidos por modelos, usando a MDA, uma apresentação gráfica da ordem das transformações e dos mapeamentos entre plataformas é necessária. Tal apresentação pode guiar o projetista de aplicações na especificação das consecutivas transformações M2M, necessárias para transformar um PIM em um PSM. O Modelo de Features é utilizado para efetuar esta transformação. No entanto, este modelo não oferece maneiras para especificar a ordem com que suas características são utilizadas para transformar um PIM em um PSM, nem para especificar mapeamentos de modelos fonte de um sistema entre plataformas alvo. Para resolver esta carência do FM, a abordagem FOMDA propõe o uso de *workflows*.

A definição de *workflow* que consta na organização *Workflow Management Coalition* [54] determina que um *workflow* é a automação de um processo de negócio ou a facilitação deste processo, de maneira total ou parcial, realizada por um sistema de software. Para a abordagem FOMDA, o uso de *workflows* define um mecanismo semi-automático para aplicar mapeamentos de transformações para modelos de um sistema. A ferramenta FOMDA Toolkit pode utilizar estes *workflows* para guiar os projetistas de aplicações na transformação de um PIM em um PSM e no mapeamento de um modelo fonte para uma plataforma alvo.

Na abordagem FOMDA, o *workflow* representa o primeiro nível de organização de transformações. Ele é definido como FOMDA *Workflow,* porque os elementos nele definidos precisam representar instâncias dos meta-dados do meta-modelo FOMDA. Neste *workflow*, as características selecionadas no FM são transformações de alto nível, especificadas como atividades. As atividades são organizadas por transições, que indicam o mapeamento de um modelo fonte para uma determinada plataforma (característica selecionada no FM).

No FOMDA *Workflow,* as transições podem conter guardas. Estas podem especificar o tipo de modelo esperado como resultado de uma transformação. Além disso, este *workflow* pode conter ações, que especificam uma intervenção do projetista de aplicações em uma determinada transformação de um modelo. Por exemplo, uma ação pode indicar ao projetista de aplicações que ele precisa selecionar um elemento do modelo de sistema e adicionar este elemento em um parâmetro de um transformador. Ainda, objetos podem ser utilizados no *workflow.* Estes servem para especificar, dentre outros recursos, parâmetros de entrada e saída de transformações.

Para cada combinação de características do PDM (FM), que determinam uma ou mais plataformas alvo, é necessário um *workflow* que documente as atividades de transformação. Isto motiva a organização de transformações em camadas (enquanto uma camada de transformação refere-se a cada transformação de uma plataforma alvo). Cada camada recebe como entrada um modelo em alto nível e aplica alguma transformação, tornando o modelo específico de uma ou mais características (transformação PIM-para-PSM). O *workflow*, assim como o PDM e os transformadores, pode ser reutilizado para transformar qualquer modelo que precise da combinação de características do PDM. A Figura 25 demonstra a organização das características selecionadas no PDM da Figura 22 em um *workflow*.

No *workflow* definido na Figura 25, as características selecionadas no PDM são identificadas por atividades decoradas com o estereótipo «*FeatureElement*» e representam transformadores de plataformas alvo em alto nível. O *workflow* precisa apresentar todas as características selecionadas no PDM, mesmo as que não são utilizadas para aplicar transformações no modelo do sistema.

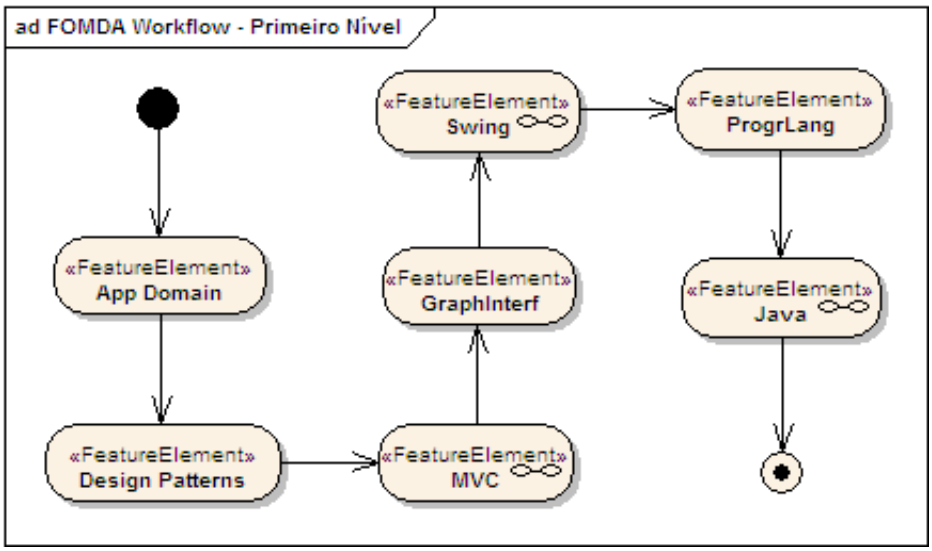


Figura 25 - Exemplo de Workflow para Transformações em Alto Nível

Como as características são utilizadas para transformar um PIM em um PSM e depois o PSM em código, é necessário informar qual delas recebe o PIM, quais geram o PSM, quais geram código e quais não efetuam transformações no modelo do sistema. Para isso, podem ser especificados estados de guarda nas transições das atividades de transformação. O valor dos estados de guarda pode ser informações, que identificam o modelo gerado pelas atividades de

transformação. Estados de guarda são utilizados com o propósito de documentação na abordagem FOMDA.

O modelo da Figura 25 pode ser melhor documentado utilizando estados de guarda nas transições das atividades de transformação. A Figura 26 demonstra o uso de guardas no *workflow*.

Na Figura 26, a transição do estado inicial para a atividade *App Domain* contém a guarda *[PIM]* e identifica em qual característica o PIM deve ser transformado primeiro. Logo, uma transição entre atividades, juntamente com uma guarda, documenta o mapeamento de um modelo do sistema para uma transformação, que é representada por uma das características selecionadas no PDM. A transição entre a atividade *App Domain* e *Design Patterns* contém a guarda *[identity]*. Esta última recebe este nome porque é usada para indicar que a atividade fonte da transição (*App Domain*) não efetua nenhuma transformação no modelo recebido como entrada. No caso da Figura 26, o PIM é o modelo de entrada para a atividade *App Domain* e o modelo de saída da mesma é a identidade do PIM, ou seja, ele próprio sem transformações.

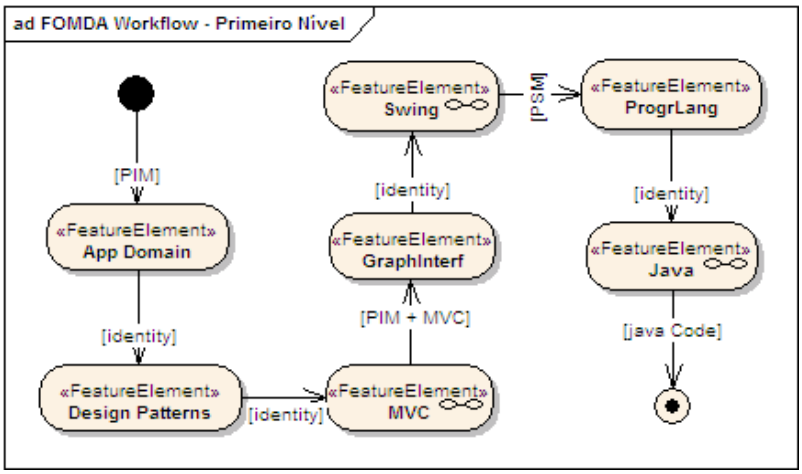


Figura 26 - Modelos Trafegados entre Atividades de Transformação

A atividade identificada como *Design Patterns* também não efetua transformações no modelo de entrada e gera como saída o mesmo modelo recebido como entrada. A atividade identificada como *MVC* recebe a saída da atividade *Design Patterns* (o PIM) e gera como saída (identificada pelo estado de guarda *[PIM + MVC]*) o PIM mais alguns elementos que

representam o padrão de projeto MVC. Estes elementos são classes decoradas com os estereótipos «*entity*», «*boundary*» e «*control*».

A transformação realizada em *MVC* é caracterizada como de nível intermediário entre um PIM e um PSM. O PSM é gerado depois que a atividade *Swing* for alcançada no *workflow*. Entretanto, esta atividade precisa do modelo gerado na atividade *MVC* para realizar sua transformação. A transformação na atividade *Swing* pode, por exemplo, fazer com que todas as classes que estejam decoradas com o estereótipo «*boundary*» (que representa uma classe de interface gráfica no padrão MVC) herdem de JPane, que é uma classe definida para interface gráfica de Swing da linguagem de programação Java. Esta é uma prática definida como modelagem intencional (Czarnecki e Eisnecker 2004), em que os modelos do sistema são especificados para gerar código para uma plataforma alvo. A atividade "Java" aplica uma transformação no PSM e gera código Java a partir dele.

A atividade *Swing* precisa de um modelo como entrada que contenha o estereótipo «*boundary*». Esta informação não foi especificada no *workflow* apresentado na Figura 26 e, com o objetivo de documentar melhor as entradas e saídas das transformações de alto nível, objetos são definidos para identificar as entradas e saídas para as atividades de transformação, como mostrado na Figura 27. Estes objetos são instâncias do meta-dado *FeatureParameter* do meta-modelo FOMDA e são identificados pelo estereótipo «*FeatureParameter*». Além disso, estes objetos podem conter *tagged values,* que documentam o que o parâmetro precisa conter, seja ele de entrada ou saída de uma característica.

Parâmetros de entrada para uma característica são identificados por uma transição que contém o estereótipo «*FeatureInput*». Tal transição precisa ter como destino uma atividade decorada com o estereótipo «*FeatureElement*» e como fonte um objeto decorado com o estereótipo «*FeatureParameter*».

Parâmetros de saída de uma característica documentam os possíveis modelos resultantes de transformações da mesma. Parâmetros de saída são identificados por transições decoradas com o estereótipo «*FeatureOutput*». Estas transições precisam ter como destino um objeto decorado com o estereótipo «*FeatureParameter*» e como fonte uma atividade decorada com o estereótipo «*FeatureElement*».

O uso dos objetos e estereótipos identificados na Figura 27 é adequado para documentar o que cada característica do PDM precisa receber como entrada e gerar como saída. Neste sentido, torna-se mais simples o mapeamento de um modelo (gerado por uma transformação em uma atividade) para outra atividade de transformação.

Estereótipos aplicados em transições identificam o estado de um modelo. O estereótipo «*SourceModel*» pode ser utilizado em transições e identifica que o estado de um modelo é de um modelo fonte.

Transições decoradas com o estereótipo «*TargetModel*» indicam que um modelo está no estado de modelo alvo. Isto determina que é possível efetuar, em modelos que se encontram neste estado, uma transformação M2C, que gera código para uma aplicação. Portanto, no exemplo da Figura 27, a transição da atividade de transformação *Swing* para *ProgrLang* identifica, além do modelo (estado de guarda *[PSM]* especificado na transição) que é trafegado de uma transformação para outra, um modelo pronto para geração de código.

Transições decoradas com o estereótipo «*IntermediateModel*» identificam modelos em nível intermediário entre um PIM e um PSM. Isto significa que o modelo trafegado por transições decoradas com este estereótipo está em um estado independente de plataforma, porém mais refinado do que o que está em um estado de modelo fonte.

Transições decoradas com o estereótipo «*SourceCode*» identificam que o estado de um modelo é o de código fonte para uma aplicação. No exemplo da Figura 27, a transição da atividade de transformação *Java* para o estado final do *workflow* indica que o estado do modelo trafegado é de código fonte. Tal transição contém o estado de guarda *[java Code]*, que indica o tipo de código gerado na atividade de transformação *Java*.

O *workflow* especificado na Figura 27 documenta as transformações em alto nível. O objetivo principal deste *workflow* é guiar o projetista de aplicações no mapeamento de um modelo fonte de um sistema para atividades de transformação, enquanto o modelo alvo não for alcançado. Isto é possível, porque foram configurados no *workflow* os tipos dos parâmetros de entrada e de saída das atividades de transformação e a ordem com que elas são usadas pelo projetista de aplicações, para aplicar transformações.

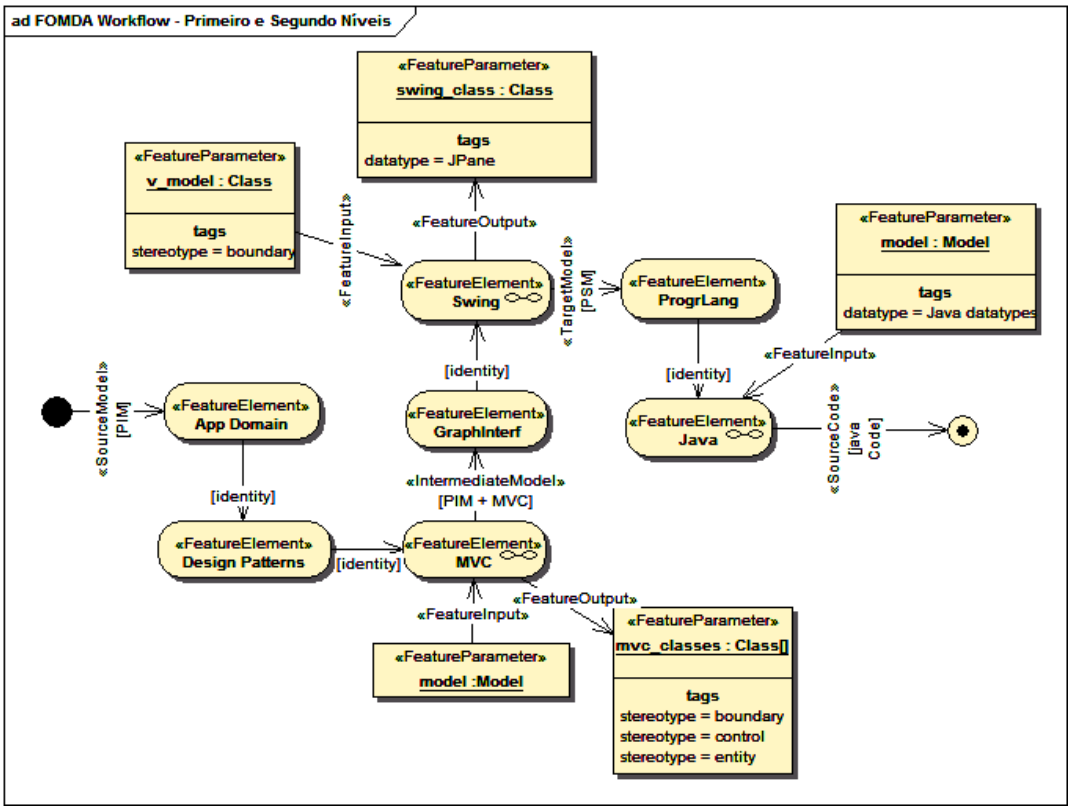


Figura 27 - Workflow para Entradas e Saídas das Transformações

O *workflow* oferece visões de transformações de alto nível, que recebem como entrada modelos fonte e geram modelos alvo. A transformação é aplicada internamente nas características do PDM (representadas como atividades decoradas com «*FeatureElement*»), pelo terceiro e quarto nível de organização de transformações. Para a abordagem FOMDA, atividades com o estado de sub-atividade contêm outro Diagrama de Atividades, que documenta as transformações internas das características.

Cada *workflow* (representado como uma atividade em um outro *workflow*) representa uma organização detalhada da transformação efetuada pela atividade de transformação. Isto quer dizer que atividades que contêm um estado de sub-atividade precisam documentar mapeamentos e transformações que são realizadas internamente nelas. A Figura 28 apresenta um exemplo, em que um *workflow* organiza as transformações internas da atividade de transformação denominada MVC.

Workflows de alto nível identificam atividades de transformação. As atividades que contêm um estado de sub-atividade contêm transformações internas. As transformações

internas de uma atividade de transformação são especificadas em um outro *workflow*, denominado de *workflow* de baixo nível. *Workflows* de baixo nível precisam ter o mesmo nome das atividades que os referenciam no *workflow* de alto nível. O *workflow* denominado MVC (especificado na Figura 28) é um *workflow* de baixo nível. Ele é referenciado pela atividade *MVC,* definida no *workflow* de alto nível mostrado na Figura 27.

O primeiro passo para especificar transformações internas de uma atividade de transformação é recuperar os parâmetros de entrada que ela possui. Como as atividades de um *workflow* de alto nível contêm parâmetros de entrada e de saída, em *workflows* de baixo nível estes parâmetros podem ser utilizados. A adição dos parâmetros de entrada e de saída de uma atividade em um *workflow* de baixo nível pode ser feita automaticamente, se os parâmetros de entrada foram definidos no *workflow* de transformações alto nível. Isto significa que os parâmetros de uma atividade de transformação podem ser trazidos para os *workflows* que identificam sub-estados de atividades. Como um trabalho futuro, pretende-se adicionar recursos na ferramenta FOMDA Toolkit, para a especificação dos *workflows*.

O *workflow MVC* contém ações que identificam tarefas que o sistema, ou o projetista de aplicações, precisa realizar para transformar um modelo fonte em um modelo alvo. No exemplo da Figura 28, a ação identificada como *select* informa que uma seleção de um elemento deve ser realizada no modelo fonte. Este modelo deve estar contido no parâmetro de entrada da característica denominado *model* (primeiro objeto do *workflow*). No mesmo exemplo, a transição da ação *select* para um objeto decorado com o estereótipo «*SourceModelElement*» (denominado *domain_class*) determina o tipo de elemento que deve ser pesquisado no modelo fonte, que está contido no parâmetro de entrada de MVC (o parâmetro *model*). O tipo do objeto *domain_class* é *mof.core.Model* e isto implica que elementos deste tipo são pesquisados no parâmetro *model*. Os objetivos de informar a seleção de elementos nos parâmetros de entrada são: mapear valores corretos para parâmetros de entrada de um transformador e possibilitar que isso ocorra de forma automatizada.

Após a seleção de um elemento em parâmetros de entrada de uma atividade de transformação, é necessário mapear o elemento para parâmetros de transformação de um transformador. Ações identificadas como *mapping* definem que o elemento selecionado precisa ser adicionado em um parâmetro de um transformador. No exemplo da Figura 28, a ação *mapping* informa que os elementos selecionados no parâmetro de entrada de uma

atividade de transformação (que são do tipo *mof.core.Class*) precisam ser mapeados para os parâmetros de transformação "*mInput*", "*vInput*" e "*cInput*". Todos estes parâmetros recebem elementos do tipo *mof.core.Class*. Isto configura um mapeamento possível, porque os tipos dos parâmetros de transformação devem ser os mesmos dos elementos que são mapeados para eles.

O estereótipo «*TransformationParameter*» deve ser utilizado em objetos e especifica um parâmetro de um transformador. Transições de objetos decorados com este estereótipo para atividades decoradas com o estereótipo «*Transformation Descriptor*», devem estar decoradas com o estereótipo «*TransformationInput*». Elas indicam que o objeto decorado com o estereótipo «*TransformationParameter*» representa um parâmetro de entrada do transformador.

No *workflow MVC* (ver Figura 28), é possível visualizar as atividades *Model Transformer*, *View Transformer* e *Controller Transformer*. As três podem ser executadas em paralelo. Isto significa que estas atividades não precisam ser ordenadas, porque as transformações que elas realizam no elemento que está contido em seus parâmetros de transformação são independentes. Atividades decoradas com o estereótipo «*TransformationDescriptor*» são descritores de transformação e podem documentar uma transformação (como comentado na Seção 7.1). Elas podem conter transições decoradas com «*TransformationAlgorithm*». Este estereótipo é usado para documentar qual é o algoritmo de transformação de um transformador.

No FOMDA *Workflow*, transformadores TMD são objetos que são decorados com o estereótipo «*Transformer*». Estes objetos precisam definir o nome e o tipo do transformador. O tipo precisa ser uma classe válida no sistema e, portanto, esta classe deve herdar da classe *AbstractTransformer* ou implementar a interface *Transformer*.

Ações identificadas como *execute* determinam a execução de um transformador. No exemplo da Figura 28, as ações *execute* determinam a execução dos seguintes transformadores: *ModelTransformer*, *ViewTransformer* e *ControllerTransformer*.

Os transformadores devem ter concluído sua transformação para gerar o modelo alvo. Isto é representado no *workflow* pelo estado de união do Diagrama de Atividades. No

exemplo da Figura 28, o modelo alvo é obtido no final do *workflow*, após a execução dos transformadores *ModelTransformer*, *ViewTransformer* e *Controller-Transformer*.

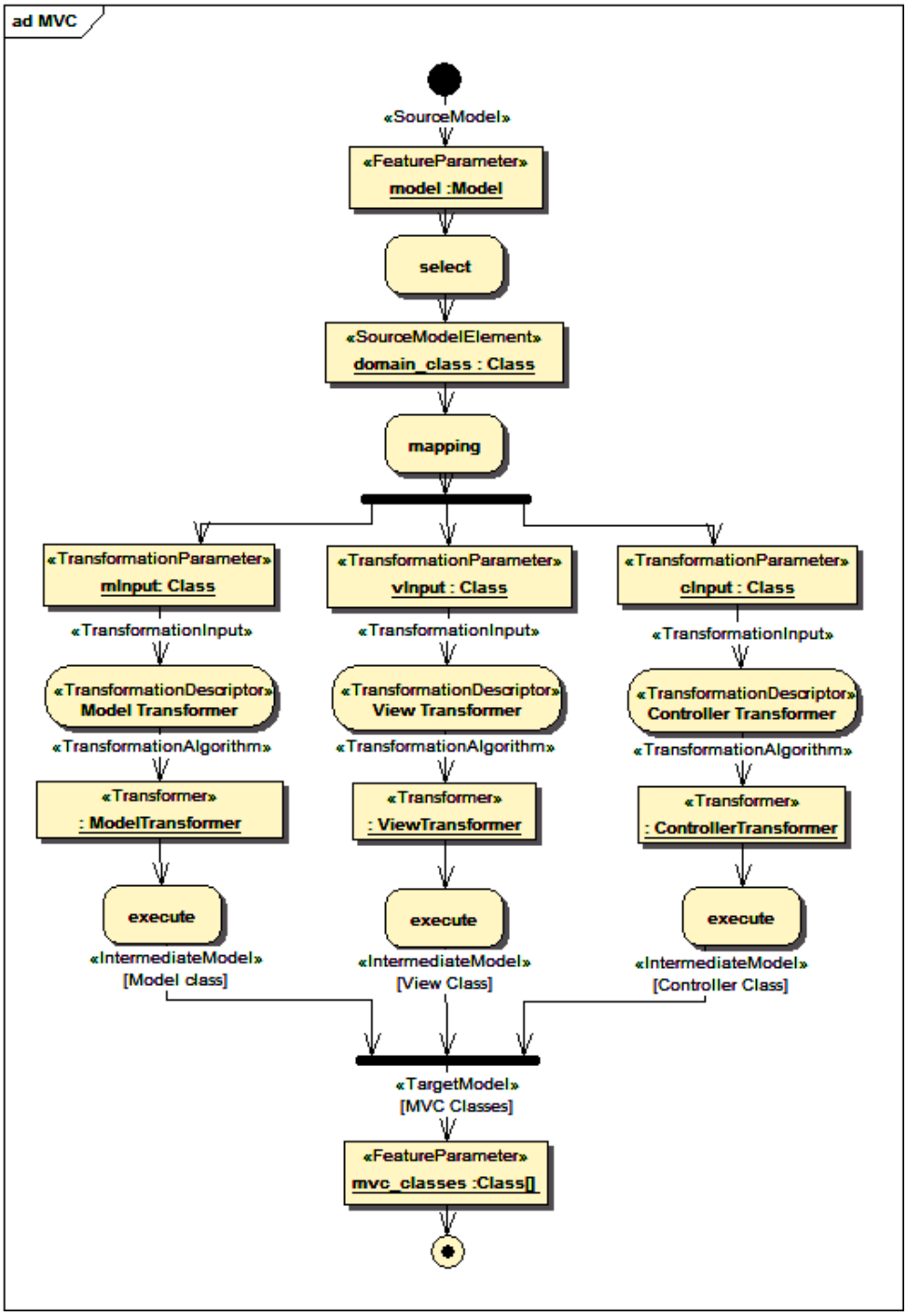


Figura 28 - MVC Workflow

8 O Protótipo de Ferramenta FOMDA Toolkit

Um protótipo de ferramenta, denominado FOMDA Toolkit, foi desenvolvido para validar a abordagem FOMDA. Os seguintes recursos são disponibilizados no protótipo: a especificação e instanciação do PDM (primeira e segunda etapa de organização de transformações); a especificação de transformadores; e a composição de transformações (terceira etapa de organização de transformações).

8.1 Recursos para a Primeira e Segunda Etapas de Organização de Transformações

As seguintes funcionalidades relacionadas com o PDM estão disponíveis no protótipo:

- Especificação do PDM (constituída de um Modelo de Features);
- Documentação das características do Modelo de Features;
- Instanciação do PDM para configurar plataformas alvo.

O PDM é um Modelo de Features, que contém características de plataformas. A Figura 29 (a) mostra os recursos definidos no protótipo FOMDA Toolkit, que podem ser utilizados pelo projetista de transformações para construir o PDM: botões para a definição dos relacionamentos do Modelo de Features (no canto superior direito da figura); um diagrama para a edição do modelo (no centro da figura); e uma representação do modelo estruturada em árvore (no canto superior esquerdo da figura).

Recursos para a documentação das características do PDM são identificados na Figura 29 (c). Estas características podem ser documentadas quanto a sua natureza (arquitetural ou funcional), categoria (documentação ou transformação) ou tempo de criação (execução ou compilação). O Diagrama de Features disponibilizado pelo protótipo FOMDA (centro da Figura 29) pode ser usado para a edição das características do PDM. Ele oferece as seguintes funcionalidades: a) seleção das características no modelo, que é conhecida como instanciação

do Modelo de Features (segunda etapa de transformação de modelos); b) edição do nome das características; c) adição de relacionamentos em uma característica do modelo; e d) organização das características no diagrama.

Em cada característica do PDM, podem ser especificadas suas regras de composição. Existem duas maneiras de especificar regras em características do PDM, na FOMDA Toolkit. Uma das formas é adicionar relacionamentos em uma característica. Isto pode ser feito utilizando os botões para edição de relacionamentos (encontrados no canto superior direito da Figura 29) ou usando o Diagrama de Features mostrado na Figura 29 (b). Os tipos de relacionamentos que podem ser especificados são aqueles explicados na Seção 3.3.2.

Composition Rules são informações textuais de uma característica e identificam a segunda forma de especificar regras para composição de uma característica no PDM. A FOMDA Toolkit oferece funcionalidades para que o projetista de transformações especifique as regras textuais para composição de uma característica. A Figura 29 (c) mostra uma aba denominada *Composition Rules,* onde tais regras podem ser definidas para uma característica do PDM. Tomando como exemplo o modelo mostrado na Figura 21, a regra textual "requires MVC" pode ser adicionada na característica *Swing* e determina que, se *Swing* for selecionada no modelo, então MVC deve ser selecionada também.

Funcionalidades para documentar as características do PDM foram adicionadas no protótipo FOMDA Toolkit. *Rationales* são documentações de uma característica, que podem ser visualizadas graficamente no Diagrama de Features. Estas especificam detalhes de uma característica, como, por exemplo, os atributos (no sentido literal) que a mesma possui. Objetivos e decisões também são documentações que podem ser adicionadas em uma característica do PDM, mas não são mostradas graficamente no Diagrama de Features. Estas últimas podem ser especificadas na aba *Issues and Decisions*.

A ferramenta FOMDA Toolkit também guia o projetista de uma aplicação durante a instanciação do PDM. Cada característica identificada na Figura 29 (b) contribui para a instanciação do PDM e pode compor uma plataforma alvo. As relações estabelecidas entre as características do FM determinam a sintaxe para a composição delas no PDM: a ferramenta FOMDA Toolkit verifica os tipos de relacionamentos estabelecidos entre as características e, de acordo com cada relacionamento, seleciona automaticamente algumas características.

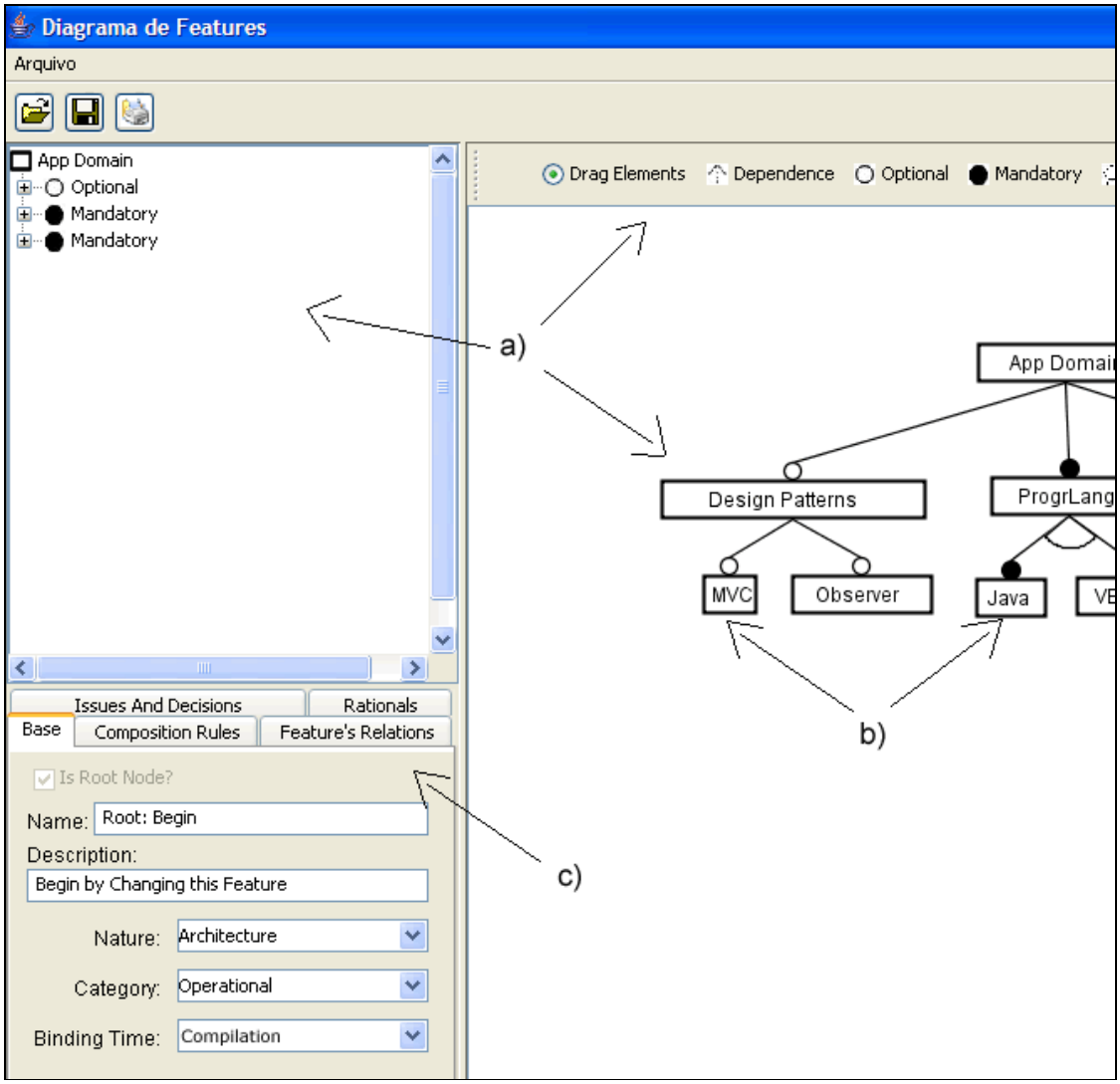


Figura 29 - Tela do Protótipo para Especificar o PDM

Sempre que uma característica for selecionada pelo projetista de aplicação, a instanciação do PDM é realizada pelo protótipo FOMDA Toolkit. Para isto, este protótipo verifica quais características podem ser selecionadas automaticamente e em quais relacionamentos é necessário que o projetista escolha uma característica no PDM. A instanciação do PDM ocorre da seguinte forma:

- Se uma característica A for selecionada e esta possuir uma relação de obrigatoriedade com uma característica B, em que A é hierarquicamente superior a B, então B é selecionada automaticamente;

- Se uma característica C for selecionada e esta for hierarquicamente inferior a uma característica B, e B for inferior a uma característica A, então as características B e A são automaticamente selecionadas (nesta mesma ordem) durante a instanciação do FM;
- Se uma característica A for selecionada e esta tiver um relacionamento obrigatório de exclusão mútua, que relaciona duas características (B e C), e estas são hierarquicamente inferiores à característica A, então a ferramenta requisita ao projetista de uma aplicação que ele escolha a seleção de B ou de C;
- Se uma característica A for selecionada e esta tiver um relacionamento "or", que identifica a necessidade de selecionar uma ou mais características relacionadas com A, e este relacionamento contém as características B e C, sendo estas últimas hierarquicamente inferiores à característica A, então a ferramenta requisita a seleção de B e/ou de C ao projetista de aplicação;
- Se uma característica A for selecionada e esta tiver um relacionamento que indica a não obrigatoriedade de outra(s) característica(s) (seja ele um relacionamento opcional de exclusão mútua ou opcional de uma única característica), então a ferramenta não utiliza este relacionamento para efetuar a seleção automática das características relacionadas;
- Se uma característica A for selecionada e esta tiver um relacionamento de dependência com uma característica B, sendo que A dependente de B, então a ferramenta seleciona automaticamente a característica B;
- Sempre que uma característica A for selecionada e esta for mutuamente exclusiva com uma característica B, por um relacionamento de exclusão mútua, e B já estiver selecionada no modelo, então o presente protótipo oferece ao projetista de aplicação a possibilidade de desmarcar a característica B.
- Sempre que uma característica A for desmarcada, então todas as características hierarquicamente inferiores a ela, que também estiverem selecionadas, são automaticamente desmarcadas.

8.2 Recursos para a Quarta Etapa de Organização de Transformações

A abordagem FOMDA define que os transformadores devem ser adicionados nas características do PDM. Transformadores devem ser escritos com a linguagem Java e compilados. Para poderem ser utilizados no protótipo FOMDA Toolkit, os mesmos devem ser empacotados em um arquivo com a extensão *.jar*. No momento, arquivos com a extensão *.jar* são anexados ao protótipo em tempo de compilação. No entanto, futuramente, os mesmos deverão ser anexados ao protótipo em tempo de execução.

Por fim, os transformadores podem ser adicionados às características do PDM, utilizando os recursos definidos na Figura 30.

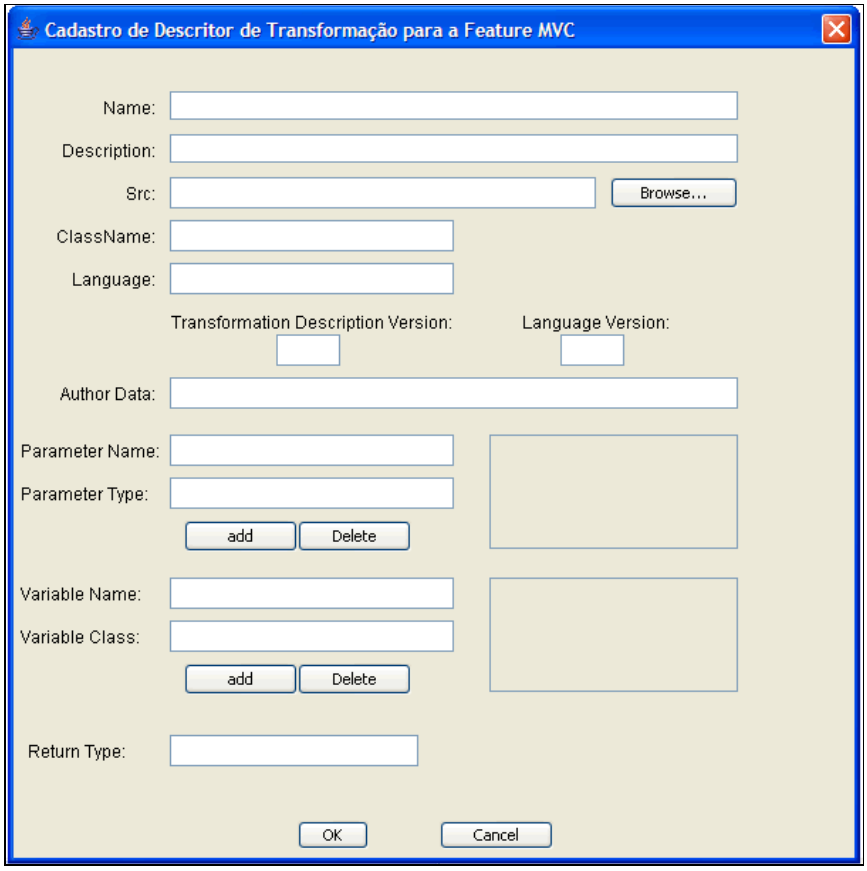


Figura 30: - Tela do Protótipo para Especificar um Descritor de Transformação

Os seguintes campos (exibidos na Figura 30) especificam, em um descritor de transformação, as seguintes informações:

- ✓ *Name (obrigatório)* - o nome do transformador.
- ✓ *Description (opcional)* - a descrição de o que o transformador faz.
- ✓ *Src (opcional)* - caso o transformador seja especificado em uma linguagem para transformação de modelos que não seja em Java, o arquivo que contém o algoritmo do transformador pode ser informado no campo identificado como *Src*. Neste caso, no campo identificado como *ClassName,* deve ser especificado o nome da classe que interpreta a linguagem utilizada para especificar o algoritmo do transformador. Se o campo *Src* estiver vazio, então o transformador é a classe informada no campo *ClassName*.
- ✓ *ClassName (obrigatório) - representa o próprio* transformador. Deve ser especificado o nome completo (a hierarquia de pacotes mais o nome da classe) de uma classe escrita em Java. Por exemplo: *com.transformadores.TransformadorDeModelos*, em que *Transformador DeModelos* é o nome da classe e *com.transformadores* é a hierarquia de pacotes desta classe. Se, ao invés de ser um transformador, a classe representa um interpretador, então a informação deste campo precisa ser classe, que é o interpretador.
- ✓ *Language (opcional)* - a linguagem utilizada para escrever o algoritmo do transformador.
- ✓ *Transformation Description Version (opcional)* - a versão do algoritmo de transformação.
- ✓ *Language Version (opcional)* - a versão da linguagem utilizada para especificar o transformador.
- ✓ *Autor Data (opcional)* - dados a respeito do autor do transformador.
- ✓ *Parâmetros de Transformação são adicionados em (opcionais):*
  - ♦ *Parameter Name (obrigatório)* - O nome do parâmetro de transformação;
  - ♦ *Parameter Type (obrigatório)* - O tipo do elemento que é armazenado pelo parâmetro;
  - ♦ *Uma tabela apresenta os parâmetros de transformação adicionados no descritor de transformação.*
- ✓ *Variáveis Compartilhadas são adicionadas em (opcionais):*
  - ♦ *Variable Name (obrigatório)* - O nome da variável compartilhada;

- ♦ *Variable Class (obrigatório)* - O tipo de elemento que pode ser armazenado na variável compartilhada;
- ♦ *Uma tabela apresenta as variáveis compartilhadas adicionadas no descritor de transformação.*

✓ *Return Type (obrigatório)* - o tipo de retorno do transformador, que deve ser um tipo válido para o protótipo.

8.3 Recursos para Terceira Etapa de Organização de Transformações

Na abordagem FOMDA, a organização de transformações na terceira etapa serve para compor transformações e também para executá-las. A composição de transformações pode ser realizada das seguintes formas: adicionando dependências de um parâmetro de um transformador pelo resultado/retorno da execução de outro transformador; adicionando dependências de um parâmetro de um transformador por uma variável compartilhada por outro; adicionando cláusulas nos transformadores, que executam transformações relacionadas com as cláusulas. A execução ocorre somente se as características que as cláusulas testam estão selecionadas no PDM. Além disso, o mapeamento de um elemento de um modelo de sistema para parâmetros de transformadores (disponibilizados em uma das características do PDM) e a execução destes por parte do projetista de aplicação, fazem parte da organização de transformações de terceiro nível.

A ferramenta FOMDA Toolkit oferece recursos para a terceira etapa de organização de transformações (apresentadas na Figura 31). A Figura 29(b) destaca no PDM a característica MVC. Para efetuar as transformações internas em MVC, é necessário selecionar, em um *pop-up* que aparece na tela quando um clique com o botão direito do mouse é feito sobre uma característica do modelo, a opção “Abrir Visão Interna da Característica”. Isto determina a abertura da tela de “visão interna de uma característica do PDM” (apresentada na Figura 31) pelo protótipo FOMDA.

O protótipo FOMDA Toolkit não oferece recursos para a definição dos parâmetros de entrada e de saída das características. No entanto, o mesmo possibilita a abertura de um modelo de sistema e a manipulação dos elementos deste último. Na Figura 31 (a) é possível visualizar uma aba denominada *"Input 1"*. Esta aba apresenta um modelo de um sistema estruturado em árvore. *Input 1* é um parâmetro da característica MVC, que é representado como uma aba na tela de visão interna desta característica.

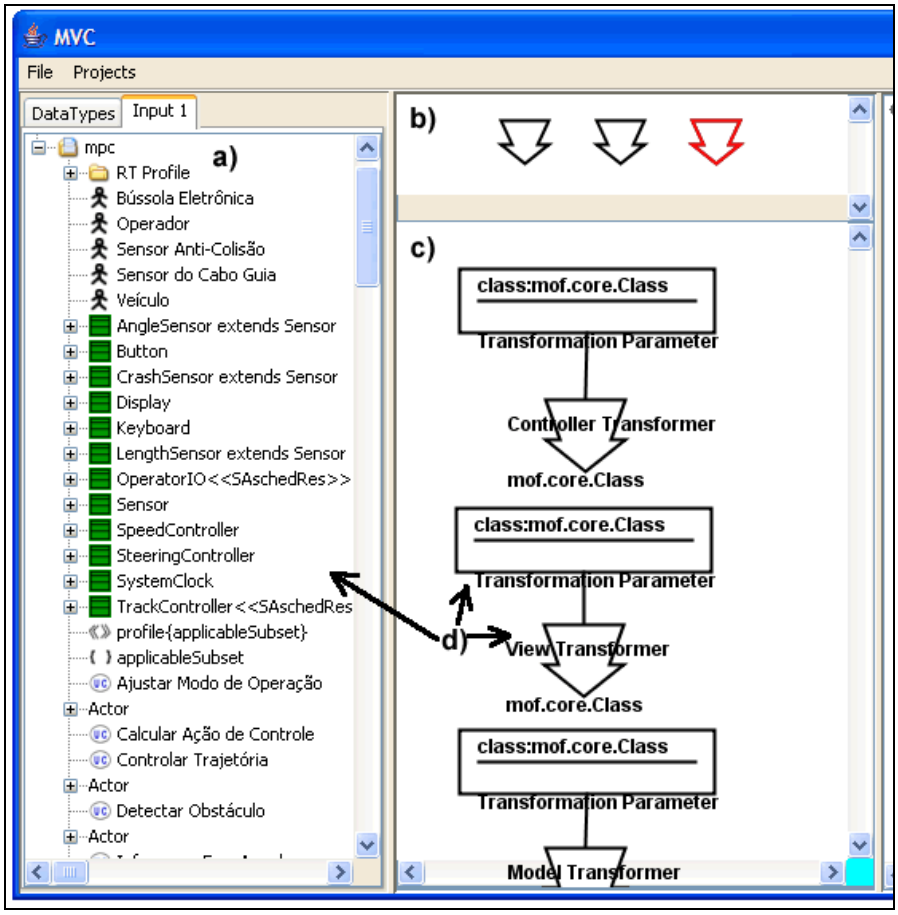


Figura 31 - Tela do Protótipo para Terceira Etapa da FOMDA

É possível adicionar mais de um parâmetro nas características do PDM. Para isto, é necessário abrir no protótipo a tela de visão interna das características. Para adicionar um parâmetro de entrada em MVC, por exemplo, é necessário selecionar (na interface mostrada na Figura 31) o menu "File" e neste a opção "Abrir Modelo Fonte". Para adicionar um modelo de um sistema em um parâmetro de uma característica, este modelo precisa ser um documento no formato XMI ou XMILight (UMT 2006). Em breve, o protótipo deve ser

estendido para a especificação dos parâmetros de entrada e de saída das características do PDM.

A Figura 31 (b) apresenta os descritores de transformações da característica MVC. Cada descritor de transformação é representado como um botão na tela, que representa a visão interna da característica que o contém. Estes botões servem para adicionar no Diagrama de Composição de Transformações DCT (Figura 31(c)) um novo transformador idêntico ao transformador representado pelo botão. Quando o mouse é posicionado sobre estes botões, as informações do nome e a descrição do transformador são mostradas. Os transformadores podem ser adicionados no DCT, com o objetivo de compor novos transformadores e de utilizar um transformador para executar uma transformação.

A Figura 31 (d) identifica recursos oferecidos para a execução de transformações. Um parâmetro de um transformador pode ser selecionado no DCT. Os elementos de um modelo, que estão em uma das entradas da característica (identificada pela Figura 31), podem ser selecionados e arrastados para o Diagrama de Composição e Transformação. Isto determina a adição deste elemento no parâmetro de transformação selecionado no DCT. O transformador é executado quando o projetista de aplicação seleciona a opção "*execute*" no *pop-up menu*. O *pop-up menu* é mostrado quando um clique com o botão direito do mouse é feito sobre um descritor de transformação (este último deve estar representado no DCT).

A Figura 32 apresenta um exemplo de composição de transformações, utilizando o DCT do protótipo. Este exemplo não tem relação com os exemplos anteriores, porque ele tem o objetivo de mostrar o que pode ser realizado para compor transformações utilizando o protótipo FOMDA Toolkit. Neste exemplo, o transformador identificado como *Schedulable Transformer* contém um parâmetro de transformação de nome *object* e uma variável compartilhada identificada com o mesmo nome. Além disso, o mesmo transformador contém duas cláusulas *"If"*. Uma destas cláusulas testa se a característica *"RTSJ"* está selecionada no PDM e a outra testa se a característica *"FemtoAPI"* está selecionada no PDM. A primeira delas determina a execução do transformador identificado como *Schedulable RTSJ Transformer* e a segunda determina a execução do *Shedulable FemtoAPI Transformer*. Ambos os transformadores dependem do valor contido na variável compartilhada por *Schedulable Transformer*.

Se o transformador *Schedulable Transformer* for executado, então a sua variável compartilhada recebe um elemento do modelo do sistema que é adicionado por este transformador. No algoritmo de transformação deste transformador, o elemento que é adicionado em sua variável compartilhada (denominada *object*) é o mesmo objeto recebido como parâmetro pelo transformador. Para compor transformações, é necessário ter conhecimento do algoritmo do transformador. A descrição deste algoritmo pode ser informada no descritor de transformação (no campo *Description*).

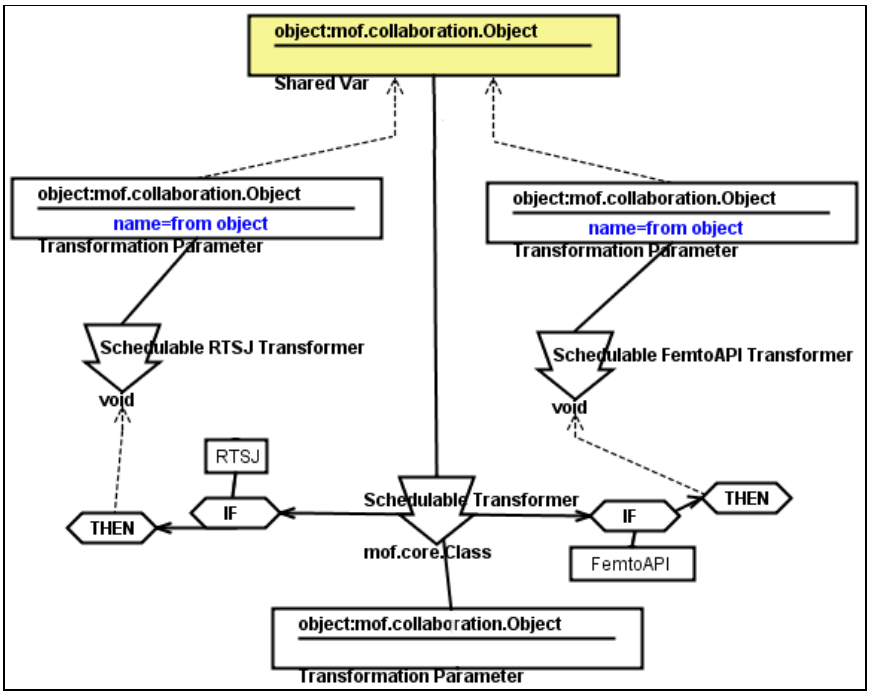


Figura 32 - Exemplo de Composição de uma Transformação na FOMDA Toolkit

Após a execução do transformador *Schedulable Transformer*, as cláusulas relacionadas a ele são testadas e os transformadores relacionados com cada uma destas cláusulas são executados. Isto ocorre se, e somente se, as características que estas cláusulas avaliam estiverem selecionadas no PDM. Como o valor do parâmetro dos transformadores *Schedulable RTSJ Transformer* e *Schedulable FemtoAPI Transformer* depende do valor armazenado pela variável compartilhada de *Schedulable Transformer*, a execução destes transformadores precisa, primeiro, recuperar o valor contido na variável compartilhada de *Schedulable Transformer*. Este valor deve ser adicionado no parâmetro de transformação dos transformadores *Schedulable RTSJ Transformer* e *Schedulable FemtoAPI Transformer*. Isto é

tratado pela FOMDA Toolkit, que organiza as "pendências" de um transformador antes de executar o seu algoritmo.

8.4 Restrições e Trabalhos Futuros

Como foi argumentado no desenvolvimento desse trabalho, a organização de transformações de terceiro nível pode auxiliar no reuso de transformadores TMD. Identificou-se que estes transformadores precisam oferecer outros recursos, além daqueles já definidos nesse trabalho, para a organização de transformações de terceiro nível. Assim, como um trabalho futuro, pretende-se adicionar aos transformadores TMD os recursos identificados na Seção 7.2.5.

No presente trabalho, um estudo de caso descreveu como transformar um modelo em alto nível (PIM) para um modelo de baixo nível (PSM). No entanto, não foi exemplificado neste estudo de caso como transformar um CIM em um PIM. Logo, pretende-se, futuramente, estender o estudo de caso apresentado aqui e demonstrar como a FOMDA pode ser usada para transformar um modelo de domínio (CIM), que contenha requisitos de sistemas de cadeira de rodas (apresentados na Seção 3.3.2), em um modelo independente de plataforma (PIM).

O presente trabalho apresentou o meta-modelo da FOMDA. Este último pode ser utilizado como um Perfil da UML, para especificar mapeamentos e transformações de modelos de sistemas. Como outro trabalho futuro, pretende-se especificar os meta-dados do meta-modelo FOMDA em um Perfil para a UML.

9 Estudo de Caso

A abordagem FOMDA pode ser utilizada para projetar qualquer tipo de sistema. Esta abordagem pode ser adaptada para o desenvolvimento de Sistemas Embarcados de Tempo Real (STRE), dado que ela oferece mecanismos para capturar requisitos de sistemas (especialmente os não funcionais) e para mapeá-los para serviços providos por plataformas. Para ilustrar esta adaptação, este Capítulo apresenta um estudo de caso relacionado com o projeto de um STRE.

9.1 Organizando as Plataformas Alvo para o Desenvolvimento de Sistemas Embarcados de Tempo Real

O projetista de transformações inicia a organização dos mapeamentos e transformações de modelos pela primeira das etapas da FOMDA. Nesta etapa, o projetista de transformações precisa especificar em um Modelo de Features as características das plataformas que são utilizadas pra desenvolver muitos tipos de STREs.

O exemplo da Figura 33 apresenta possíveis requisitos não funcionais para um STRE. Ela define que qualquer produto deve ter pelo menos as seguintes características: uma linguagem de programação (Java, C, ou C++); um fluxo de execução que pode ser concorrente ou não; uma abordagem para administração de tempo que pode ser feito com programação de baixo nível (*low level programming*) ou por chamadas de sistemas (*system calls*); e um processador de hardware relacionado (PowerPC, DSP, FemtoJava, ou x86). A característica de infra-estrutura é opcional, mas quando *concurrent* ou *system call* é selecionada, então a infra-estrutura é necessária. Conseqüentemente, se infra-estrutura é necessária, o projetista pode escolher entre os sistemas operacionais, API e *middleware*.

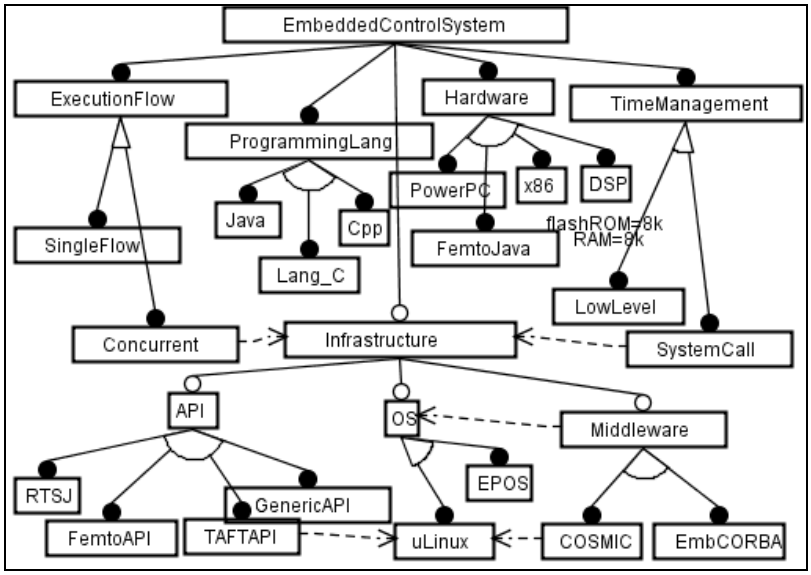


Figura 33 - Exemplo de um PDM para Desenvolvimento de Sistemas Embarcados

Regras de composição textuais devem ser definidas na abordagem FOMDA para especificar as relações de dependência e de exclusão mútua como regras para seleção de plataformas. A Tabela 3 apresenta algumas regras textuais para a composição das características do FM exemplificado na Figura 33.

Tabela 3 - Composition Rules para STRE

Característica	Composition Rules
FemtoAPI	requires “Java” requires “FemtoJava”
RTSJ	requires “Java” excludes “Cpp” and “Lang_C”
TAFTAPI	requires “Lang_C” excludes “Java” and “Cpp”
FemtoJava	excludes “Cpp” and “Lang_C” and “TAFTAPI” and “”GenericAPI”
LowLevel	excludes “API”
API	excludes “LowLevel”

Depois de definir o FM, o projetista de transformações pode configurar a plataforma alvo selecionando as características relacionadas. Esta seleção é conhecida como instanciação do Modelo de Features (Oliveira et al. 2004). A Figura 34 apresenta uma possível instanciação do modelo apresentado na Figura 33. As características selecionadas estão marcadas em cinza. Neste exemplo, as seguintes características estão selecionadas: Java, como linguagem de programação; fluxo de execução concorrente, administração de tempo

usando chamadas de sistemas; FemtoAPI como uma infra-estrutura de API; e FemtoJava como processador.

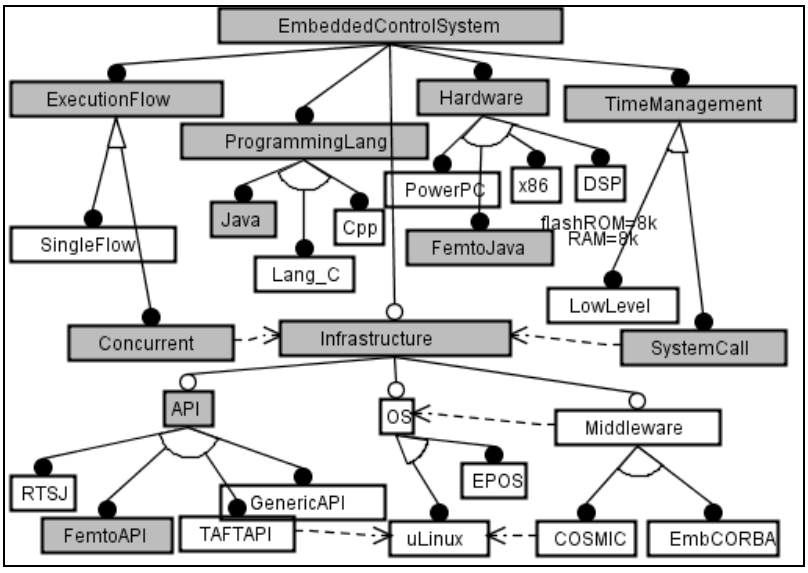


Figura 34 - Instanciação do Modelo de Features para SETR

9.2 Organizando as Características Selecionadas do PDM

Depois de instanciar o PDM é possível utilizá-lo para aplicar transformações em um modelo fonte. Este modelo fonte é descrito nesta seção, que apresenta um estudo de caso que exemplifica a geração de um SETR usando as etapas de organização da abordagem FOMDA. O problema consta na transformação de um modelo de sistema especificado em alto nível para um modelo específico de FemtoAPI. O modelo em consideração contém requisitos para um sistema de controle de movimento de uma cadeira de rodas motorizada (Wehrmeister et al. 2005). Estes requisitos estão decorados com restrições de tempo real, que são marcações definidas pelo *Profile for Schedulability, Performance, and Time (SPT)*(Becker et al. 2002, Wehrmeister et al. 2005). A Figura 35(a) apresenta dois objetos colaborando (extraídos de um diagrama mais geral) que são usados para exemplificar os mapeamentos propostos.

Os objetos na Figura 35(a) foram selecionados para este estudo porque eles contém informações temporais importantes nas quais a respectiva implementação é altamente

dependente de plataforma. Os seguintes aspectos, pelo momento, devem ser observados durante a implementação do sistema e conseqüente transformação de modelo para código: administração de tempo, execuções concorrentes e periódicas, e também controle de *deadline*. Portanto, o presente trabalho é desafiado pelo seguinte problema: como gerar o SETR final de acordo com a seleção de plataformas pelo projetista da aplicação? Essa seção guia o leitor na abordagem FOMDA para resolver o problema referido.

As características selecionadas na Figura 34 são utilizadas para aplicar transformações em um modelo de um sistema. Isto quer dizer que um modelo (um PIM) precisa ser transformado para a plataforma alvo composta de: Java, FemtoJava, Concurrent, SystemCall e FemtoAPI. Cada uma destas características pode transformar um PIM em um PSM.

A Figura 35(b) dá um exemplo de organização de algumas das características selecionadas na Figura 34 para aplicar transformações no modelo fonte da Figura 35(a). A transição do estado inicial do *workflow* para a atividade de transformação denominada *EmbeddedControlSystem* identifica o início da transformação. O estereótipo «*SourceModel*» pode ser utilizado em transições e identificam que o estado de um modelo é de um modelo fonte. Transições podem conter estados de guarda, que identificam que tipo de modelo uma transição trafega para os demais elementos do diagrama. Atividades de transformação são decoradas com o estereótipo «*FeatureElement*» e precisam conter como nome uma das características selecionadas no MF.

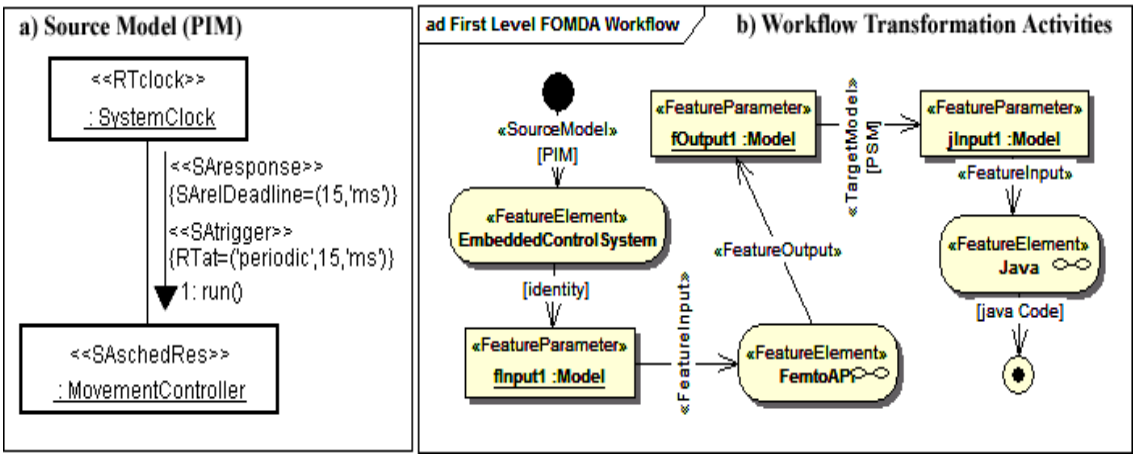


Figura 35 - Modelo Fonte e o Workflow para Organizar Transformações

Nesse exemplo, a transformação inicia com o mapeamento (transição) de um PIM (estado de guarda), que é o modelo-fonte (estereótipo «*SourceModel*»), para a atividade de transformação *EmbeddedControlSystem*. Em seguida, uma transição que contém o estado de guarda *identitiy* informa o mapeamento de um resultado desta atividade para o objeto denominado *fInput1*.

O objeto *fInput1* definido na Figura 35 b) está decorado com o estereótipo «*FeatureParameter*». Isto indica que ele é um parâmetro da característica que determina o nome da atividade de transformação. Ele contém uma transição decorada com «*FeatureInput*» para a atividade de transformação *FemtoAPI*. Então, este objeto é um parâmetro de entrada para a característica *FemtoAPI*, visto que esta é representada no *workflow* como uma atividade que contém o estereótipo «*FeatureElement*».

Um parâmetro de entrada de uma característica indica o tipo do elemento de um modelo de um sistema que pode ser mapeado para a característica. No exemplo da Figura 35(b), o objeto *fInput1* é do tipo *Model*, o que significa que a característica *FemtoAPI* precisa receber como entrada um elemento deste tipo para aplicar transformações. Do contrário, FemtoAPI não pode realizar a transformação.

De acordo com a ordem das transformações identificadas nesse exemplo, a atividade de transformação FemtoAPI recebe no parâmetro de entrada *fInput1*, o modelo retornado pela atividade EmbeddedControlSystem. Este modelo é descrito como *identity*. Esta atividade efetua uma transformação e armazena o resultado no seu parâmetro de saída *fOutput1*. O resultado desta transformação no modelo de entrada é identificado pelo estado de guarda contido na transição do objeto *fOutput1* para o objeto *jInput1*. No exemplo, o tipo de modelo trafegado pela transição entre estes objetos é o PSM. Todo o mapeamento que é realizado no *workflow* de alto nível deve ser feito entre um parâmetro de entrada e um parâmetro de saída. Por esse motivo o modelo que é saída da atividade FemtoAPI não foi mapeado diretamente para o parâmetro de entrada de java (objeto *jInput1*). Além disso, existe a possibilidade de mapear a saída de FemtoAPI para outras atividades de transformação, o que justifica a necessidade de uma mapeamento ser realizado entre um parâmetro de saída e, no mínimo, um parâmetro de entrada de uma outra característica.

No exemplo da Figura 35(b), o elemento de um modelo contido no parâmetro de saída *fOutput1* é um elemento pronto para geração de código. A transição do objeto *fOutput1* para o *jInput1* indica o mapeamento do elemento de modelo, contido no parâmetro *fOutput1*, para o parâmetro de entrada da atividade de transformação Java, denominado *jInput1*. Java aplica uma transformação que gera código para uma aplicação. Isto é identificado pela transição desta última para o estado final do *workflow*. Tal transição contém o estado de guarda *java code*.

No mesmo exemplo da Figura 35(b) o *workflow* documenta a ordem com que as características do MF são usadas para aplicar transformações (M2M e M2C). Além dessas especificações, é necessário identificar como as atividades de transformação transformam um modelo contido em um de seus parâmetros de entrada. Isto é realizado na FOMDA definindo *workflows* de baixo nível.

Um *workflow* de baixo nível é definido por uma atividade de transformação (do *workflow* de alto nível) que contém um estado de sub-atividade. A Figura 36 documenta as transformações internas da atividade de transformação *FemtoAPI*. O objeto *fInput1* é o mesmo do *workflow* da Figura 35(b). Existe uma transição dele para uma ação identificada como *select*. Esta determina que uma seleção (no parâmetro de entrada da atividade de transformação) de elementos do mesmo tipo dos objetos do modelo fonte (elementos do modelo da aplicação do projetista de aplicações) decorados com o estereótipo «*SourceModelElement*».

Dentro de um parâmetro de entrada do tipo *Model*, uma pesquisa é realizada para selecionar apenas elementos do tipo *Object*. Quando um elemento é selecionado, ele é mapeado (ação identificada como *mapping*) para parâmetros de transformação. Estes parâmetros são identificados por objetos decorados com o estereótipo «*TransformationParameter*» e são utilizados por transformadores da categoria TMD (quarto nível de organização de transformação).

Após mapear elementos do tipo *Object* para o parâmetro de transformação, identificado como *Object*, do transformador *sptTransf*, o projetista pode requisitar a execução deste transformador. Isto é documentado no *workflow* com uma ação denominada *execute*. A execução do transformador gera como saída um modelo identificado pela guarda FemtoAPI

PSM, ou seja, um PSM que contém características de FemtoAPI. A sub-atividade FemtoAPI terminou, mas o projetista precisa seguir a transição dela para a próxima atividade ou objeto, identificado no *workflow* da Figura 35(b).

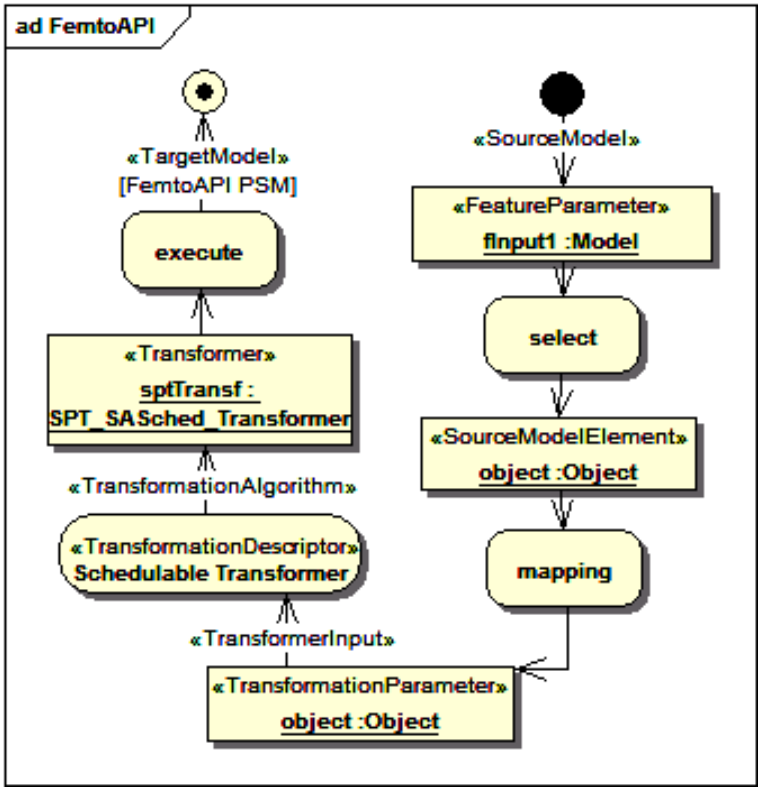


Figura 36 - FemtoAPI Workflow

9.3 Aprimorando o Estudo de Caso

No âmbito de identificar o *workflow*, algumas questões são relevantes: a) como identificar os parâmetros de uma transformação? b) como organizar transformações e seus parâmetros (levando em conta a mudança de plataformas alvo no MF? c) como reutilizar os modelos gerados no caso de haver mudança de plataformas alvo? Estas questões podem ser respondidas analisando corretamente o MF.

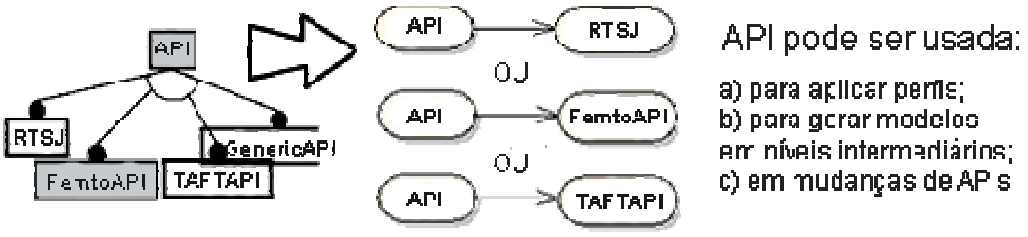


Figura 37 - Troca de Plataforma Alvo e Mudança no Workflow

A Figura 37 apresenta um exemplo em que as características API e FemtoAPI estão selecionadas no MF. O propósito com que FemtoAPI é utilizada para aplicar transformações em um modelo é converter classes deste modelo, especificadas em uma visão PIM, em classes onde os tipos dos atributos são de tipos definidos pela API FemtoAPI. Além disso, algumas das classes especificadas na visão PIM precisam, após a transformação ser realizada em FemtoAPI, herdar da classe *RealtimeThread*.

É necessário identificar o que é o PIM para a característica FemtoAPI e, dessa maneira, identificar um pré-requisito para ela. No trabalho publicado anteriormente (Basso, Oliveira e Becker 2006), optou-se por especificar que a entrada de FemtoAPI é um modelo decorado com marcações do perfil SPT (Becker et al. 2002), ou seja, um modelo de nível intermediário entre um PIM e um "*FemtoAPI PSM*". O mesmo pré-requisito é mantido para a característica FemtoAPI no presente trabalho.

Para as características RTSJ, TAFTAPI e GenericAPI o parâmetro de entrada é o mesmo que para FemtoAPI. O propósito que elas serão usadas num contexto de transformação de modelos também é o mesmo que o de FemtoAPI. Logo, uma conclusão que pode ser tirada em relação às quatro características filhas de API é que todas elas recebem como parâmetro de entrada o mesmo modelo decorado com o perfil SPT. Isto garante que caso exista a troca de FemtoAPI por RTSJ, pode-se reutilizar o mesmo parâmetro de entrada de FemtoAPI em RTSJ.

Dado que todas as APIs precisam receber como entrada o mesmo modelo, é preciso garantir que isto ocorra num processo de transformação de modelos na abordagem FOMDA. Uma análise detalhada do MF (mostrado na Figura 33) revela que a característica API pode determinar um modelo de saída, que serve como entrada comum para RTSJ, FemtoAPI, TAFTAPI e GenericAPI. Isto é possível porque a característica API é hierarquicamente

superior a estas últimas e, portanto, pode ser dita como sendo de nível mais abstrato. Logo, características de nível mais abstrato podem ser utilizadas para reusar modelos em caso de troca nas plataformas alvo de um sistema.

Com base nessa análise, foi possível concluir que características de nível hierárquico mais alto no Modelo de Features podem ser úteis para reuso de modelos, em caso de mudanças de plataformas alvo. Isto pode ser observado no centro da Figura 37, em que três partes de três *workflows* descrevem a mudança de plataformas alvo no quesito API. API é, portanto, uma característica do FM que define como saída, um mesmo modelo que é usado como parâmetro de entrada pelas características FemtoAPI, RTSJ, TAFTAPI e GenericAPI.

O modelo que é parâmetro de saída de API pode ser reutilizado, sem alterações, em caso de mudança das características relacionadas com API. A mudança da característica FemtoAPI para RTSJ é representada no MF pela seleção desta última. Sendo assim, o reuso de modelos pode ser garantido em níveis de independência que são determinados pela hierarquia das características no MF. Em caso de mudança de plataformas alvo, é possível recorrer para modelos independentes de plataforma (ou de nível intermediário entre o PIM e o PSM). Tais modelos podem ser encontrados em transformações realizadas em características de níveis hierárquicos superiores à característica que representa a plataforma alterada.

Outra análise das características do MF identificou que algumas delas não têm relação hierárquica, mas podem contribuir para a reutilização de modelos. Isto determina que a solução para reuso de modelos (apresentada anteriormente) não se aplica nos casos em que as características não possuem relação hierárquica. Tomando como exemplo a Figura 33, as características selecionadas, que são usadas com o propósito de aplicar transformações em um modelo de um sistema e que não tem relação hierárquica são: Java, Concurrent, FemtoAPI, SystemCall e FemtoJava. Todas elas precisam estar documentadas no *workflow* de alto nível. Analisando os propósitos de cada uma, é possível avaliar se elas efetuam transformações M2M, gerando modelos intermediários entre um PIM e um PSM ou mesmo o PSM, e M2C, gerando código para uma aplicação.

A análise dos propósitos das características do FM implica na ordenação das transformações no FOMDA Workflow. No exemplo, Java pode ser uma característica que tem o propósito de gerar código Java e, portanto, tem como parâmetro de entrada um modelo

contendo classes. Concurrent é uma característica relacionada com concorrência e pode identificar no *workflow* a geração de um modelo que contenha marcações relacionadas com concorrência. É possível utilizar o perfil SPT para decorar elementos de um modelo e especificar neles as características relacionadas com concorrência. Da mesma maneira, SystemCall pode identificar que marcações que representam chamadas de sistema precisam ser especificadas num modelo de sistema.

Características como Concurrent e SystemCall são exemplos de transformações que identificam a necessidade de especificar marcações, definidas por perfis da UML, num modelo de sistema. Tais marcações precisam especificar detalhes não funcionais em um PIM, tornando-o específico destes detalhes. Um modelo que contém marcações não funcionais pode ser definido como um PIM ou de nível intermediário entre o PIM e o PSM. Ele pode ser reutilizado em transformações para mais de uma plataforma alvo. FemtoAPI e RTSJ são exemplos de transformações que podem receber estes tipos de modelos como entrada e, dessa maneira, tais modelos podem ser reutilizados na mudança de plataformas alvo.

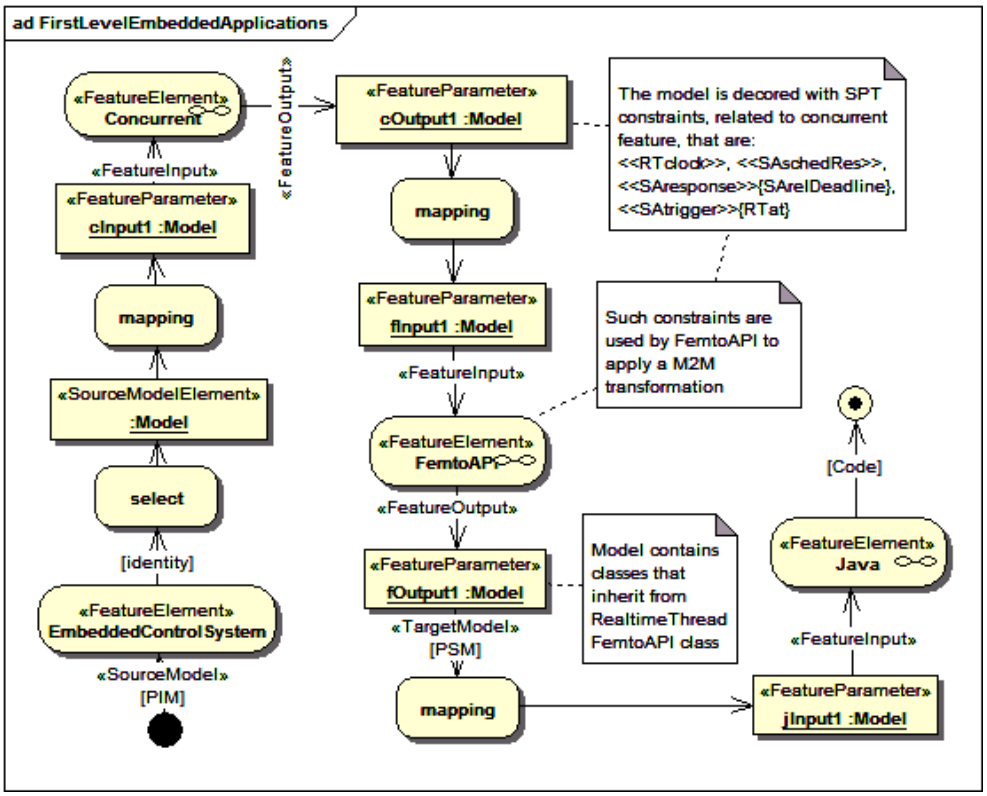


Figura 38 - Workflow que Documenta os Pré-requisitos das Transformações

Com base nas análises realizadas no MF da Figura 33, foi possível especificar os *workflows* definidos na Figura 38. O *workflow* denominado *FirstLevelEmbedded Applications* oferece a vantagem de, em caso de mudança de plataforma como, por exemplo, mudar de FemtoAPI para RTSJ, que apenas a atividade FemtoAPI e seus respectivos parâmetros sejam removidos do *workflow*. Neste último devem ser inseridos a nova atividade RTSJ e seus parâmetros de entrada e de saída. Além disso, esta atividade precisa ser configurada em um outro *workflow* para aplicar transformações internas.

O *workflow* definido na Figura 39, denominado Java, especifica as transformações internas da atividade de transformação Java do primeiro *workflow* da Figura 38. Este *workflow* especifica que elementos do tipo *mof.core.Class* são selecionados no parâmetro de entrada da característica Java. Em seguida estes parâmetros são mapeados para o parâmetro de transformação do transformador *Generic Java Transformer*. Por fim, este transformador é executado e o código Java, respectivo à classe mapeada para o parâmetro deste transformador, é gerado.

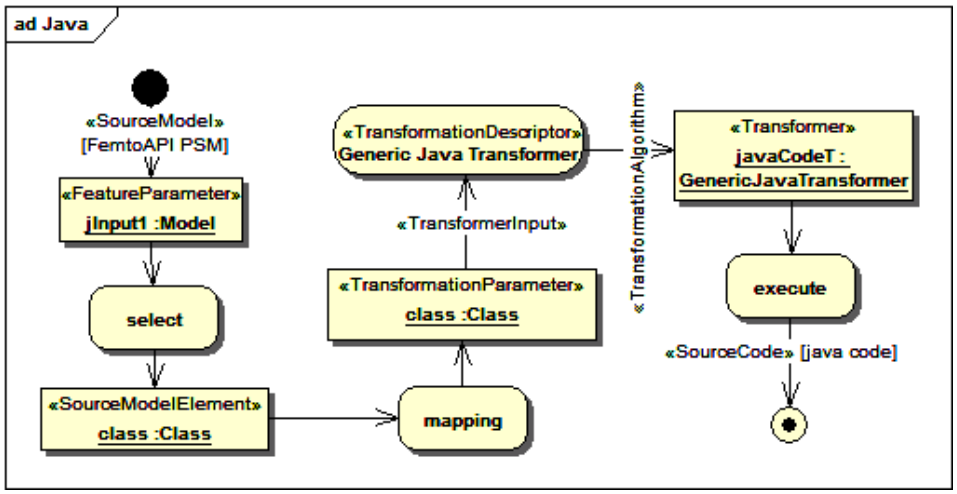


Figura 39 - Java Workflow

O *workflow* que documenta a atividade de transformação *Concurrent* é mostrado na Figura 40. Os transformadores identificados nesta figura auxiliam o projetista na especificação das marcações do Perfil SPT em elementos de modelos de sistemas. Estes transformadores podem oferecer *wizards* ou até mesmo Linguagens de Modelagem de Domínios Específicos, conhecidas como *Domain Specific Modeling Languages* (DSML). A vantagem em acrescentar transformadores neste tipo de característica está em auxiliar o

projetista de aplicações na especificação das marcações nos modelos de um sistema, caso o mesmo já não as tenha especificado.

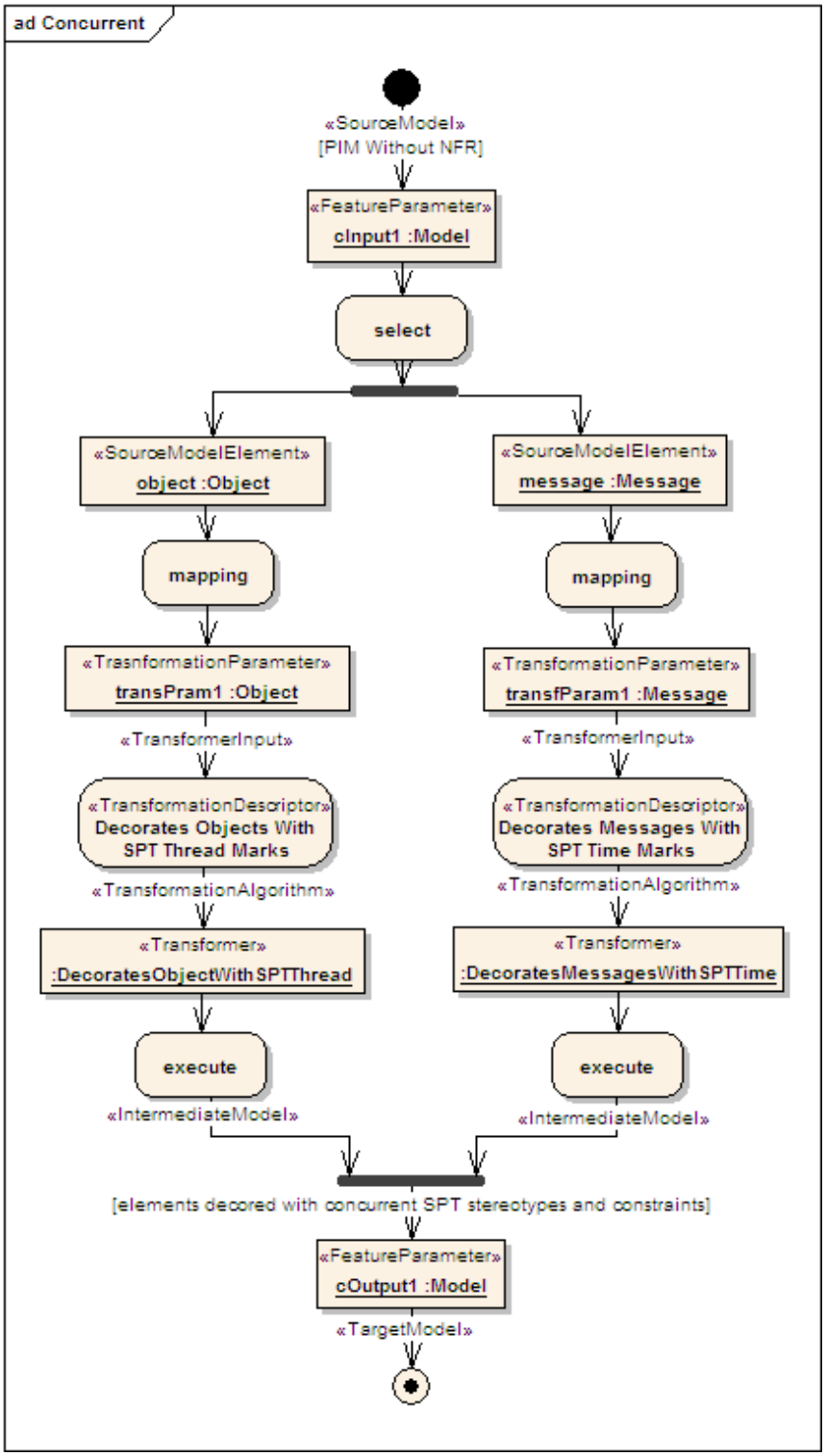


Figura 40 - Concurrent Workflow

A organização das transformações de um PIM para um PSM é documentada nos *workflows* identificados neste estudo de caso. Eles foram especificados de modo a responder as questões levantadas anteriormente. Para isso, foram feitas análises detalhadas dos relacionamentos entre as características definidas no Modelo de Features e do propósito que cada uma é utilizada para aplicar transformações em um modelo de entrada. Como resultado da análise, foi identificada a necessidade de modelos de nível intermediário estarem decorados com marcações definidas por perfis da UML. Logo, algumas das características selecionadas no MF identificam a necessidade do uso de perfis para a transformação de modelos. Modelos decorados com marcações podem ser reutilizados em caso de uma mudança das plataformas alvo.

9.4 Organizando Transformações de Baixo Nível

Esta Seção exemplifica como utilizar transformadores de baixo nível na abordagem FOMDA. Para isso, o modelo da Figura 41 é utilizado como exemplo. O modelo da Figura 41 apresenta um Diagrama de Colaboração que é utilizado para exemplificar os mapeamentos definidos na Figura 38.

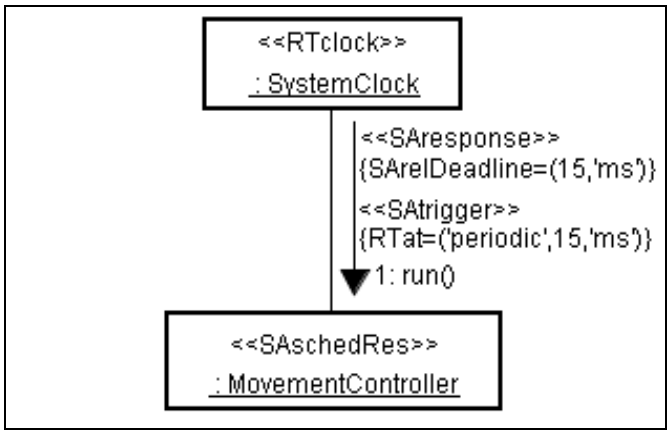


Figura 41 - Sistema de Controle de Movimento de Cadeira de Rodas

No *workflow* de alto nível da Figura 38, o modelo gerado pela transformação em Concurrent deve ser mapeado para a característica FemtoAPI. O *workflow* de alto nível foi especificado desta maneira por dois motivos: 1) porque FemtoAPI é uma característica

selecionada no PDM; 2) porque FemtoAPI, como um compilador de modelos, pode gerar um modelo que é dependente da plataforma FemtoAPI a partir do modelo da Figura 41. O *workflow* que documenta as transformações internas de FemtoAPI é definido na Figura 36 e determina os seguintes mapeamentos e transformações: i) no parâmetro *fInput1* de FemtoAPI devem ser procurados elementos do tipo *mof.collaboration.Object* (os objetos do modelo da Figura 41); ii) os elementos encontrados devem ser mapeados para o parâmetro de transformação do transformador *Schedulable Transformer*; iii) este transformador é, por fim, executado. O algoritmo de transformação deste transformador precisa verificar as marcações de um objeto (recebido como entrada no parâmetro de transformação) e transformar a classe que identifica este objeto em uma classe que contenha os mapeamentos definidos por Becker et al. (Becker et al. 2002).

Para demonstrar a transformação do modelo fonte da Figura 41, uma transformação M2M é aplicada usando o transformador da Figura 42. O transformador precisa receber como um parâmetro um elemento do tipo *mof.collaboration.Object* e, baseado no perfil SPT, deve aplicar a transformação de um modelo fonte para o modelo alvo FemtoAPI. Analisando o código do transformador, na linha 5 é posto o código para obter o parâmetro de transformação *object* (este é o elemento de modelo de sistema que é pesquisado pela ação *select* do workflow da Figura 36). A linha 7 mostra como obter a classe alvo de um objeto, baseado na especificação do MOF. A classe alvo é o elemento do modelo fonte que a transformação M2M é aplicada, tornando-a dependente da plataforma alvo FemtoAPI. A Linha 10 mostra como o transformador FOMDA testa se alguma característica do FM está selecionada. Quando a característica *Concurrent* esta selecionada no FM ou a classe alvo é ativa ou ela contém o estereótipo «*SAschedRes*», então a classe alvo precisa herdar da classe *RealtimeThread*. Este mapeamento é baseado no trabalho de Becker et al. (Becker et al. 2002).

Das linhas 18 à 22 o projetista de transformações precisa escrever um algoritmo para aplicar transformações sobre as mensagens decoradas com o estereótipo «*SAresponse*». Mensagens que contém este estereótipo são mapeadas para a plataforma alvo FemtoAPI adicionando duas operações na classe alvo: *mainTask* e *exceptionTask*. As linhas 23-43 contém o algoritmo para adicionar a operação *mainTask* à classe alvo, em que o corpo da operação (onde é determinado o código de implementação para esta operação) pode ser gerado no código da linha 31 até a 41. Este código pode ser usado para adicionar código Java

específico de FemtoAPI no elemento associado, e pode ainda ser usado posteriormente por um outro transformador. Neste algoritmo de transformação o foco está em realizar transformações M2M para modificar uma classe e não para gerar código desta.

```
1  public class SPT_SASched_Transformer extends AbstractTransformer {
2   protected mof.core.Class targetClass;
3   protected mof.collaboration.Object obj;
4   public Object doTransformation() throws Exception {
5    TransformationParameter tp1 = observer.getParameter(“object”);
6    this.obj = (mof.collaboration.Object) tp1.getElement();
7    this.targetClass = (mof.core.Class) obj.getType();
8    if (targetClass == null) throw new
       Exception(“Parameter object must contains a type”);
9    Stereotype saschedulable = obj.getAssignedStereotype("SAschedRes");
10   if (isFeatureSelected(“Concurrent”) ||
         targetClass.isActive() || targetClass.
          containsStereotype(“SAschedRes”)){
  /*the target class must inherit from RealtimeThread class*/
11     Class superClass = new
         Class(targetClass.getRootParent());
12    superClass.setName("RealtimeThread");
13    targetClass.getRootParent().addElement(superClass);
14    targetClass.getSuperClasses().add(superClass);}
15    searchForResponseStereotype();
16    return null;}
17  private void searchForResponseStereotype() {
18         …
19     createMainTaskOperation();
20     createExceptionTaskOperation();
      }
21  private void searchForSatriggerStereotype(){
22
    }
23  private void createMainTaskOperation() {
24     Operation op = new Operation();
25     op.setName(“mainTask”);
26     op.setVisibility(“public”);
27     op.setScope(“instance”);
28     op.setAbstract(false);
29     Parameter ret = new Parameter(op);
30     ret.setName(“Return_”+op.getName());
31     ret.setKind(“return”);
32     DataType voidDT = new DataType(op.getRootParent());
33     voidDT.setName(“void”);
34     op.getRootParent().addElement(voidDT);
35     ret.setType(voidDT);
36     op.addElement(ret);
37     Comment mTaskCode = new Comment(op.getRootParent());
38     mTaskCode.addAnnotatedElement(op);
39     String strCode = “”;
40     …/* designer must write algorithm to define the code
                of  the“mainTask” operation */
41      mTaskCode.setBody(strCode);
42     op.addComment(mTaskCode);
43     targetClass.addElement(op);
      }
  }
```

Figura 42 - Transformador para a Plataforma Alvo FemtoAPI

As classes alvo, pelo momento, não estão prontas para a geração de código porque a operação *mainTask*, assim como outros elementos das classes alvo, precisam conter seções de

código dentro do corpo de seus elementos, como realizado por Becker et al. (Becker et al. 2002) e por Wermesiter et al. (Wehrmeister et al. 2005). Estas seções de código nos corpos dos elementos, como para a operação *mainTask*, podem vir de muitas outras marcações em modelos fontes e cada seção de código precisa ser organizada para gerar o código correto. A solução para isso é adicionar estas seções como comentários associados com os elementos. Depois, estes comentários podem ser organizados dentro do corpo dos elementos em um transformador M2C para, então, gerar o código correto.

Das linhas 21-22 o projetista de transformações pode escrever o algoritmo para pesquisar por mensagens decoradas com o estereótipo «*SAtrigger*». O algoritmo precisa adicionar uma instância do tipo *mof.core.Attribute* na classe alvo. A restrição {*RTat = ('periodic',15,'ms')*} do estereótipo «*SAtrigger*» na Figura 41 indica que muitos atributos contendo tipos específicos de FemtoAPI precisam ser gerados (por exemplo: PeriodicParameters, AbsoluteTime, etc) e inicializados, também seguindo o mapeamento definido em Becker et al. (Becker et al 2002) e Wehrmeister et al. (Wehrmeister et al.).

10 Conclusões

O presente trabalho apresenta a abordagem Features-Oriented Model-Driven Architecture (FOMDA) como uma solução para mapeamento de transformação de modelos de sistemas em um MDD. Esta abordagem é proposta para auxiliar projetistas de aplicações na definição de mapeamentos e transformações de quaisquer modelos de sistemas (especificados em UML) para múltiplas plataformas alvo. Isto é feito configurando transformações model-to-model e model-to-code, em camadas. Cada camada representa alguma plataforma alvo para a qual o sistema precisa ser mapeado e transformado. A tarefa de configurar mapeamentos e transformações de modelos de sistemas é dividida em quatro níveis, a fim de simplificar e tornar mais claras as transformações.

A abordagem FOMDA define que transformações de baixo nível são caracterizadas como de manipulação direta de modelos. Nesta categoria, o projetista precisa escrever algoritmos de transformadores, para aplicar transformações nos elementos do modelo fonte. Isto é feito em conjunto com a organização de transformações de terceiro nível. O segundo nível de transformação serve para configurar entradas, saídas e transformadores de plataformas alvo. Finalmente, o primeiro nível de organização de transformações representa *workflows,* que servem para guiar o projetista de aplicações nos mapeamentos e transformações em plataformas alvo.

A divisão das transformações em quatro níveis possibilitou que os transformadores pudessem ser reaproveitados no caso de troca das plataformas alvo. Quando é necessário o uso de outras plataformas, além das selecionadas no PDM, é necessária apenas uma redefinição das transformações (especificadas em alto nível no FOMDA *Workflow*). Esta tarefa se torna simples, porque estas transformações já estão especificadas nas plataformas, sendo necessário apenas que o projetista organize-as no *workflow* e informe quais são os parâmetros de entrada e saída de cada transformador.

Um estudo de caso foi apresentado para avaliar a aplicação da abordagem FOMDA no desenvolvimento de um sistema embarcado de tempo real. Este estudo apresentou como converter um modelo geral em alto nível (PIM) para um modelo específico de uma plataforma

alvo (PSM). Além disso, este estudo possibilitou avaliar o reuso de modelos e de transformações. Análises feitas no Modelo de Features e nas transformações (que podem ser realizadas pelas características desse modelo) identificaram interessantes práticas que viabilizam o reuso de modelos.

As características de nível hierárquico superior no FM podem auxiliar na identificação do uso de marcações em modelos de sistema. Um modelo decorado com marcações provenientes de perfis UML pode ser reutilizado por muitos transformadores, para diferentes plataformas. Isto determina que modelos podem ser especificados em camadas, que determinam o seu nível de independência de plataformas. Dessa maneira, é possível identificar quando um modelo pode ser reutilizado, em caso da troca das plataformas utilizadas para desenvolver uma aplicação.

Os resultados obtidos são otimistas e levam a conclusão de que a abordagem FOMDA pode auxiliar no desenvolvimento de software para múltiplas plataformas. Além disso, esta abordagem pode ser utilizada para desenvolver software para muitos domínios, como para o domínio de sistemas comerciais e de sistemas embarcados. Ela pode levar os projetistas a repensarem os seus processos de desenvolvimento de software. Estes processos podem ser decompostos em níveis de independência das características específicas das plataformas alvo, utilizadas para desenvolver sistemas.

Referências

[1] ALMEIDA, J. P. et al. **Platform-independent Modelling in MDA**: supporting abstract platforms. In: Model-Driven Architecture: Fundations and Applications 2003, University of Twenty, Holanda, 2004. pp 217-231.

[2] APPUKUTTAN, B. et al. **A model driven approach to model transformation**; In: Model-Driven Architecture: Fundations and Applications 2003, University of Twenty, Holanda, 2003. pp 1-12.

[3] BASSO, F. P. **Um Estudo Sobre o Estado da Prática no Desenvolvimento de Sistemas Embarcados**. Trabalho Individual (Mestrado – Ciência da Computação). Pontifícia Universidade Católica do Rio Grande do Sul. Porto Alegre, RS, Brasil. 2004.

[4] BASSO, F. P. **A MDA Como um Auxílio no Desenvolvimento de Sistemas Embarcados**. Trabalho Individual (Mestrado em Ciência da Computação) - Pontifícia Universidade Católica do Rio Grande do Sul, PUCRS. Porto Alegre, RS, Brasil. 2004.

[5] BASSO, F.B; OLIVEIRA, T. C.; BECKER, L. B. **Using the FOMDA Approach to Support Object-Oriented Real-Time Systems Development**. In: 9th IEEE International Symposium on Object and Component Oriented Real-Time Distributed Computing, 2006, Gyeongju. Proceedings of the 9th IEEE International Symposium on Object and Component Oriented Real-Time Distributed Computing. Los Alamitos, USA . **Anais IEEE Computer**, 2006. pp. 374-381

[6] BECKER, L. B.; HÖLTZ, R. H.; PEREIRA, C. E. **On Mapping RT-UML Specifications to RT-Java API: Bringing the Gap**; In: International Symposium on Object and Component Oriented Real-Time Distributed Computing 2002, Crystal City, USA, 2002. pp. 348-355.

[7] BETTIN, J. **Model-Driven Software Development an emerging paradigm for Industrialized Software Asset Development**. Disponível em: <http://www.softmetaware.com/ mdsd-and-isad.pdf> Acesso em: 06/03/2006.

[8] BOAS, G. E. **From the Workfloor**: Developing Workflow for the Generative Model Transformer. In: 2nd OOPSLA on Generative Techniques in the Context of MDA, 2003.

[9] BOCH, G.; RUMBAUGH, J; JACOBSON, I. **The Unified Modeling Language: User Guide**. Addison-Wesley- Longman, 1999.

[10] BRISOLARA, L. B. **Estudo do uso da UML para Modelagem de Sistemas Embarcados**. Trabalho Individual II (Mestrado em Ciência da computação), Universidade Federal do Rio Grande do Sul, UFRGS, Fevereiro, 2004.

[11] CZARNECKI, K. **Generative Programming: Principles and Techniques of Software Engineering Based on automated Configuration and Fragment-Based Component Models**. 1998. Tese (Doutorado em Ciência da Computação), Technische Universität Ilmenau, Ilmenau, Alemanha,1998.

[12] CZARNECKI, K.; SIMON H.; **Classification of Model Transformation Approaches**. In: OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. 2003. pp 1-17.

[13] CZARNECKI, K.; EISENECKER, U. **Generative Programming: Methods, Tools, and Applications**. United States of America: Addison-Wesley, 2004. 832 p.

[14] CZARNECKI, K.; ANTKIEWICZ, M.; Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Generative Programming and Component Engineering: 4th International Conference, GPCE 2005. Tallinn, Estonia. 2005. pp 1-16.

[15] DEELSTRA, S. et al. **Model Driven Architecture as Approach to Manage Variability in Software Product Families**. In: Model-Driven Architecture: Fundations and Applications 2003, University of Twenty, Holanda, 2003. pp 109-114.

[16] DOUGLASS, B. P. **Real Time UML, Developing Efficient Objects for Embedded Systems**. United States of America, Massachusetts: Addison Wesley, 1998.

[17] **Eclipse IDE - Documentação do IDE Eclipse**. Disponível em: <http://www.eclipse.org>. Acesso em: 06/03/2006.

[18] FONTOURA, M. C. **A Systematic Approach to Framework Development**. 1999. Teste (Doutorado em Ciência da Computação) – Departamento de Ciência da Computação, Pontifícia Universidade Católica do rio de Janeiro, PUC-Rio, 1999.

[19] FRANKEL, D. S. Model Driven Software Development. **Business Process Trends Journal in MDA**. 2004.

[20] GARLAN, D; SHAWN, M. **Software Architecture**: Perspectives on an Emerging Discipline. USA, Upper Saddle River: Prentice Hall, 1996.

[21] GEÓRGIA, M.; JAELSON, B. **Towards a Goal-Oriented Requirements Methodology Based on the Separation of Concerns Principle**. In: VI International Workshop on Requirements Engineering. Piracicaba, SP. 2003. pp 223-239.

[22] GRAAF, B.; LORMANS, M.; TOETENEL, H. **Embedded Software Engineering**: The State of the Practice. In: IEEE Software, Delft Univ. of Technol., Holanda, 2003. pp 61-69.

[23] GREEN, P.N.; EDWARDS, M. D. **The Modelling of Embedded Systems Using HASoC**. In: 2002 Design, Automation and Test in Europe Conference and Exhibition. Paris, França, 2002. pp 752-759.

[24] GREENFIELD, J.; SHORT, K. **Software Factories**: Assembling Applications with Patterns, Models, Frameworks and Tools; In: Third International Conference, SPLC 2004, Boston, USA, 2004.

[25] GRISS, M. L.; FAVARO, J.; D'ALESSANDRO, M. **Integrating Feature Modeling with the RSEB**. In: 5th International Conference on Software Reuse. Victoria, Canadá, 1998. pp. 76-85.

[26] GRISS M. L. **Product-Line Architectures**: Book Component-Based Software Engineering: Putting the Pieces Together. USA: Addison-Wesley, 2001. 818p. Capítulo 22.

[27] **Jamda: The Java Model Driven Architecture 0.2.4**. Disponível em: <http:// sourceforge.net/projects/jamda/>. Acesso em: 06/03/2006.

[28] **Jude - Java and UML Development Environment**. Disponível em: <http://jude.change-vision.com/jude-web/index.html > Acesso em: 06/03/2006.

[29] KANG, K. C. et al. **FORM**: A feature-oriented reuse method with domain-specific architectures. **Annals of Software Engineering, V5**. Balzer Science Publishers, 1998. pp. 143-168.

[30] **MDA - Object Management Group MDA Specifications**; Disponível em: <http://www.omg.org/mda/specs.htm>. Acesso em: 06/03/2006.

[31] MELLOR, S. J. Make Models be Assets. **Periódicos: Communications of the ACM**. Novembro 2002. Vol 45. No 11. pp 76-78.

[32] MILLER J., MUKERJI J.; **MDA Guide Version 1.0.1**: Document number: omg/2003/06/01. Disponível em: <http://www.omg.org/docs/omg/03-06-01.pdf>. Acesso em: 06/03/2006.

[33] **MOF - Object Management Group; Meta Object Facility (MOF) 2.0 Core Specification**. Disponível em: <http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-04.pdf>. Acesso em: 06/03/2006.

[34] **NetBeans IDE – Ferramenta NetBeans**. Disponível em < http://www.netbeans .org/products/ide/index.html >. Acesso em 06/03/2006.

[35] **OCL - Object Management Group: UML 2.0 OCL Specification**. Disponível em: <http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf> Acesso em: 06/03/2006.

[36] OLIVEIRA, T. C. **Uma Abordagem para Instanciação de Frameworks Orientados a Objetos**. 2001. 165 p. Tese (Doutorado em Ciência da Computação) - Pontifícia Universidade Católica do Rio de Janeiro, PUC-Rio, Rio de Janeiro, Brasil, 2001.

[37] OLIVEIRA, T. C. et al. **Enabling Model Driven Product Line Architectures**. In: Second European Workshop on Model Driven Architecture (ECMDA-FA); Canterbury, Inglaterra, 2004.

[38] **OptimalJ- Model-driven development for Java**. Disponível em: <http://www.compuware.com/products/optimalj>. Acesso em: 06/03/2006.

[39] PHILIP, J.; KOOPMAN, J. **Embedded Systems Design Issues (the Rest of the Story)**. In: International Conference on Computer Design (ICCD 96), Austin, Texas, USA, 1996. pp 310-317.

[40] **QVT-Partners - Object Management Group: QVT-Partners; Revised submission for MOF 2.0 Query / Views / Transformations RFP; Version 1.1**; Disponível em: <http://qvtp.org/>. Acesso em: 06/03/2006.

[41] RIEBISCH, M. et al. **Extending Features Diagrams With UML Multiplicities**; In: Integrated Design and Process Technology, IDPT-2002; United States of America, 2002.

[42] SELIC, B; RUMBAUGH, J. **Using UML for Modelling Complex Real-Time Systems**. Rational: Withe paper, 1998.

[43] SELIC, B. **On Software Platforms, Their Modelling with UML 2, and Platform-Independent Design**. In: .8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. Seattle, USA, 2005. pp. 15-21.

[44] TARPH, A. L. The impact of fourth generation programming languages; **ACM SIGCSE Bulletin**; Volume 16, 1984. pp 37-44.

[45] TEKINERDOGAN, B.; BILIR, S.; ABATLEVI, C. **Integrating Platform Selection Rules in the Model Driven Architecture Approach**. In: Model-Driven Architecture: Fundations and Applications 2003, University of Twenty, Holanda, 2004. pp 184-200.

[46] **Together - Borland Together Technologies**. Disponível em: <http://www.borland.com/us/products/together/index.html>. Acesso em: 06/03/2006.

[47] **UMT- UML Modeling Transformation Toolkit**. Disponível em: <http://umt-qvt.sourceforge.net> Acesso em: 06/03/2006.

[48] **XMI - Object Management Group: XML Metadata Interchange (XMI) Specification**. Disponível em: <http://www.omg.org/cgi-bin/apps/doc?formal/03-05-02.pdf>. Acessado em: 06/03/2006.

[49] **XSLT - W3C, XSL Transformations (XSLT) version 1.0.** Disponível em: <http://www.w3c.org/TR/xslt>. Acesso em 06/03/2006.

[50] ZEHUA, L. **Xdoclet Tutorial**. Disponível em: <http://www.cais.ntu.edu.sg/ ~liuzh/xdoclet_tutorial/ >. Acesso em: 06/03/2006.

[51] ZHU, Q.; MATSUDA, A.; SHOJI, M.; **An Object-Oriented Design Process for system-on-Chip using UML**. In: International Symposium on System Synthesis (ISSS '02); October 2-4, Kyoto, Japan, 2002. pp 249-254.

[52] WAGNER, F; CARRO, L. **Sistemas Computacionais Embarcados**. Anais do XXIII Congresso da SBC Jornada de Atualização em Informática (JAI). Campinas, SP, 2003.

[53] WEHRMEISTER, M.; BECKER, L.B.; PEREIRA, C.E. **Applying the SEEP Method in the Design of a Real-Time Embedded Control System for a Motorized Wheelchair**. In:

10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Catania, It, 2005.

[54] WILLINK, E. D. **UMLX:** A graphical transformation language for MDA. In: Model-Driven Architecture: Fundations and Applications 2003, University of Twenty, Holanda, 2003. pp 13-24.

[55] **Workflow Management Coalition.** Disponível em: <http://www.wfmc.org>. Acessado em: 06/03/2006.

Anexo A - Using the FOMDA Approach to Support Object-Oriented Real-Time Systems Development

Using the FOMDA Approach to Support Object-Oriented Real-Time Systems Development

Fabio Paulo Basso[1], Toacy Cavalcante Oliveira[1], Leandro Buss Becker[2]

[1)] *Faculty of Informatics, PPGCC, PUCRS, Brazil, {fbasso;toacy}@inf.pucrs.br*
[2)] *DAS/CTC/UFSC, Brazil, lbecker@das.ufsc.br*

Abstract

This paper tackles the problem of ever changing embedded systems non-functional requirements, specially the architectural ones. It proposes a solution based on Features Model and MDA standards, which is called Features-Oriented Model-Driven Architecture (FOMDA). This proposal can be used to help application designer in defining the mappings and transformations of UML models to as many target platforms as wished. This is done by configuring model-to-model and model-to-code transformations over tiers, where every tier represents some target platform properties that the system must be mapped and transformed to. To validate the proposal a case study related to the development of an embedded real-time system is presented, detailing how to transform a generic high-level UML model to a model specific for a given target platform. Obtained results are optimistic and conclude that the FOMDA approach can make designers re-think their current development process to make it more decoupled from a specific target platform.

Keywords: platform-independent modeling, platform mappings, embedded real-time systems

1 Introduction

Designing new generations of embedded real-time systems (ERTS) is a complex task, but it can become manageable by using the object-oriented paradigm and its related technologies. The Model Driven Architecture (MDA) [14], for instance, is an OMG initiative to help developers manage software development complexity using models at higher levels of abstraction. The key aspect in this technology is allowing the design of models that are decoupled from their target platform. According to Selic [1] the term platform can be understood as "any set of hardware or software mechanisms that enable execution of software applications". Nevertheless, an eminent problem of MDA is the absence of a formalism to model the possible target(s) platform(s) and, moreover, the bindings between the application model and the platform (see [1]).

A suitable formalism that has been used by system designers to model platforms is the Features Model (FM) [13][45]. It is composed by features (representing the requirements) and by textual information as composition rules and rationales (which state feature properties) [13].The idea is to define features (functional, architectural, technological, mixed, etc) from a particular domain and, based on the features relations, use it to configure different platforms.

Given that the FM is attractive to specify platforms and architectures, and that MDA is attractive to help in transformation, this work aims to provide a solution to use these approaches to compose and organize transformations based on features and consequently facilitating the mapping of a more general model for multiple platforms. Therefore, we propose the extension of the FM to support MDA's concepts and consequently define a new approach named FOMDA (Features-Oriented Model-Driven Architecture). We also designed a tool to help the application of FOMDA in the transformation of UML models.

Although FOMDA can be used to design any kind of system it is also well suited for ERTS development, given that it offers mechanisms to capture system's requirements, specially the non-functional ones (e.g. timing constraints), and map them to the services provided by the platform. To illustrate this suitability, a case study related to an ERTS design is presented in the paper.

The remainder of the paper is organized as follows. Section 2 details the main concepts from both key technologies used in this work: MDA and FM. Section 3 describes the FOMDA approach and section 4 presents the case study. The related works are presented in section 5 and the conclusions are presented in section 6.

2 Background on MDA and FM

This section details both technologies used in the scope of the current work: the Model Driven Architecture (MDA) and the Features Model (FM).

2.1 Model Driven Architecture

MDA can be understood as a "philosophy" that guides systems development. It does not define explicitly the diagrams that should be used and neither the required transformations among the three different kinds of models

that it defines: Computational Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM). All these models, or visions, may be used as system's development tiers [7][45], where each tier can be transformed into another representative model that can be used to generate specific code to some architecture. The model, in general, represents elements using the UML [9], therefore they can be represented using the Meta Object Facilities (MOF) [13].

Czarnecki et al. in [12] makes a classification of MDA transformation approaches and divides them in two categories: Model-to-Code (M2C) and Model-to-Model (M2M) approaches. In M2C transformations, the goal is to generate code directly from models or templates. In M2M transformations, the objective is to transform the input model into another model. For example, a PIM could be transformed to a PSM by adding programming languages, specific data types or stereotypes. M2M transformations are very useful when the domain platforms are heterogeneous because designers can decouple the transformations and models views on representative features. More information about M2M transformations can be found in [7][45][12].

2.2 Features Model

Features Model (FM) [13][29] is proposed to specify composition rules of a system's functional and non-functional features. It is composed by features representing requirements and by features relations representing their composition. Textual composition rules are defined by each feature and may express compositions that can not be specified only using feature relations. Tekinerdogan et al. in [45] proposed an approach to specify platforms selection rules. Moreover, platform selection rules can be represented by features relations and by textual composition rules.

The Czarnecki Features Model representation [13] defines the following relations between features (see Figure 1): (a) optional relation, used when the related feature is optional; (b) mandatory relation, used when the related feature is mandatory; (c) dependency relation, used when a feature is dependent of another feature; (d) mandatory mutually exclusive relation, used when designer must choose one feature over a set; (e) optional mutually exclusive relation, used when designer can decide between choosing one feature over a set; (f) or relation, used when at least one feature of a set is needed; "zero or more" relation, used when a set of features are optional (it is similar to Figure 1 (f) but without the filled circles).

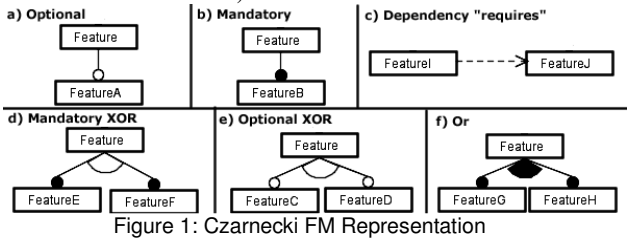


Figure 1: Czarnecki FM Representation

The FM example from Figure 2 shows possible non-functional requirements for an ERTS. It defines that any generated product should have at least the following features: a programming language (Java, C, or C++), an execution flow that can be concurrent or not; a time management approach that can be done by low level programming or by system calls; and the related hardware processor (PowerPC, DSP, FemtoJava, or x86). The infrastructure feature should be optional, but when concurrent execution flow or system call is selected then it becomes also necessary. In summary: an optional relation can become mandatory if some selected feature is dependent on it. Consequently, if infrastructure is necessary, designer must reason about the operating system, API, and middleware.

Textual composition rules should be defined in the FOMDA approach to specify dependency and mutual exclusion as platform selection rules. For example, when "LowLevel" feature is selected, then API cannot be selected. This rule may be specified into "LowLevel" feature as "excludes API" composition rule. Also, if "FemtoJava" feature is selected then "FemtoAPI" must be used as an API, and vice-versa. This rule is represented as "requires FemtoAPI" composition rule inside "FemtoJava" feature and as "requires FemtoJava" composition rule inside "FemtoAPI" feature. For simplicity, not all dependencies from the diagram are shown in the figure.

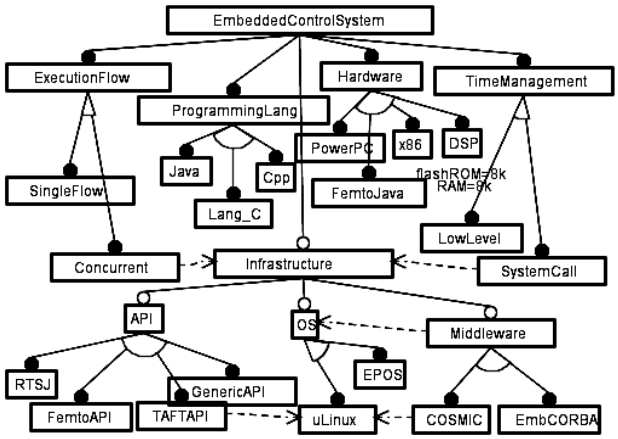


Figure 2. Example of an Embedded System's Features Model

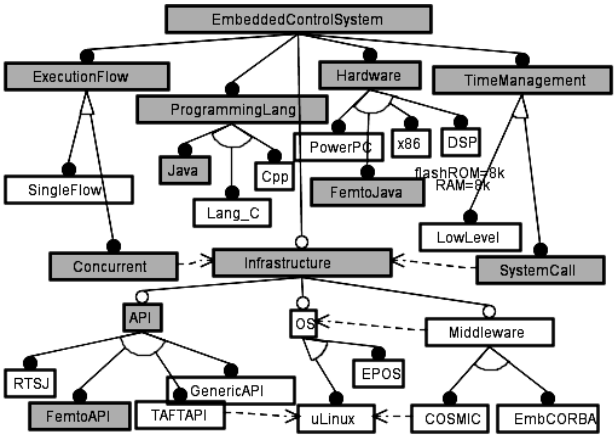


Figure 3. Features Model Instantiation

After defining FM, the designer can configure the target platform by selecting the related features. This selection is

known as instantiation [17][16], the term coming from the FORM approach [29], and Figure 3 shows a possible instantiation from the model presented in Figure 2. Observe that the selected features are marked in gray. In this example, the following features are selected: Java, as programming language; concurrent execution flow; time management designed using system calls; FemtoAPI as API infrastructure; and FemtoJava as hardware (processor). It is important to observe that this configuration can be reused by any system that needs such technologies and architectures.

There is no approach that addresses the use of features to transform a system's functional requirements (normally a UML model) on its implementation to the many architectures and technologies identified in the diagram. This motivates the proposed approach, which uses FMs together with MDA's concepts to generate the application code based on the selected features, as detailed in the next section.

3 The FOMDA Approach

3.1 Overview

Features describe systems' characteristics and can dictate how transformation occurs. For instance, features can be mapped to transformations as shown in Figure 4 (a) and (c). FM instantiation, previously described in section 2.2, provides necessary features to develop a system, subjected to the usage of transformations as shown in Figure 4 (b) and (c). However, to organize transformations across the features identified in the FM, designer must reason about the following questions: (i) where is the starting point?; (ii) what is the expected result?; (iii) what is the next transformation? The Features-Oriented Model-Driven Architecture (FOMDA) aims helping transformation designers to make a plan, design transformations and specify workflows that aims the answer of such questions, related to the Figure 4 (b), (c) and (d).

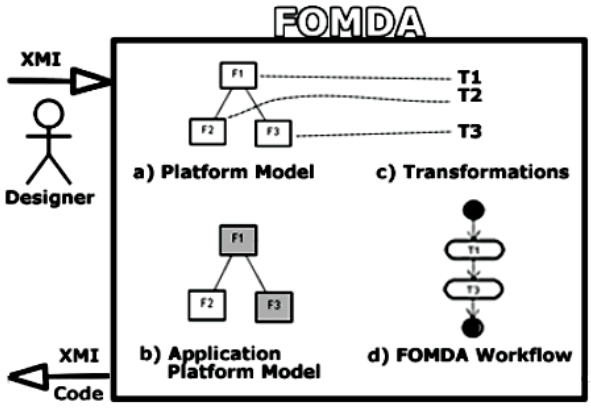


Figure 4. FOMDA Overview

The planning involves identifying the required steps to transform a high-level model (CIM/PIM) into source code. Nevertheless, generating code directly from the high-level model is not a suitable solution [12]: designers should consider first working with M2M transformations. Therefore it is required to document all the transformation tiers, informing what is done in high-level to have a clear notion of what needs to be done before some transformation finishes, together with the expected low-level output.

In the FOMDA approach the transformations configurations are organized in four levels. The first level represents a high-level transformation where the designer configures transformations, such as: the target platform specifying features from the FM used for transformation (second level); the order in which transformations are applied (transformers are the forth level); the mappings of a source model to the transformers and the composition of transformers in runtime (third level).

3.2 FM and Transformations

The features from the FM that are used in transformation are analog to target platforms in MDA [45], considering that only non-functional features are defined. Therefore our intention is to use FMs as a generic solution to apply transformations. This is the second level to organize transformations in FOMDA approach.

In the FOMDA approach the FM was extended to apply transformations: features of FM can have input and output parameters, referred as features parameters, and can be configured to apply internal transformations by adding to it, transformers.

As features may apply transformations, designers must configure the model types (or elements) that features of FM can use by configuring its input parameters. In this level, designers can constraint the source models that may be transformed by the related features. Also, designers can specify features outputs, as features parameters, indicating the types of model elements that a feature can generate after applying some transformation. The addition of input and output parameters into features serves for transformation documentation purposes.

Features can have internal transformation. This is possible because features are composed of transformers. Transformers can receive source model elements as input and than apply M2M or M2C transformations (here is where the transformation really occurs). Transformers may be composed to apply a more specific transformation or to aggregate additional functionalities. This is the third level of transformation. Finally it is required the transformation algorithm itself that must be written by the transformer designer in the so called fourth level of transformation, also known as low-level transformation.

3.3 Low-Level Transformations

The FOMDA transformers are based in the Direct-Manipulation Transformation (DMT) from Czarnecki et al [12], which is a kind of M2M transformation where the user needs to subclass some abstract classes that make transformations and apply their algorithms to manipulate directly input source models. Examples of tools that operate with DMT are Jamda [6] and UMT [15][19].

However, FOMDA transformers present some slight modifications.Firstly its transformers can receive parameters, referred as transformation parameter, eliminating the need to search inside the source model for some element, thereby the designer can make a direct mapping of model element to

transformer. Secondly its transformers can share variables with other transformers during runtime. The last difference is related with the FM, stating that a transformer will only be executed when its related feature is selected.

3.4 FOMDA Workflow

Intermediate level transformations such as consecutive PIM-to-PSM transformations require a transformation planning. This planning must document the transformations between a source model and a target model and the order that they are executed. Because FOMDA is a four level transformation approach, designer can make such planning according to the transformation level. When modeling a high-level transformation planning, platforms input and output as well as the model mapping between platforms and the order in which platforms are used for transformation should be defined. The mappings of source model elements to transformation parameters, transformers, etc, is the most relevant part of the low-level transformation planning.

A graphical view of transformation ordering and the mappings between the platforms is needed to guide the designer to apply consecutives M2M transformations. As FM does not offer ways to specify ordering, and it is not possible to specify mapping between target platforms, we proposed a workflow to guide such transformations. Consequently, activities and objects need to be created to represent what has to be done to transform a high-level model into an intermediate target model. In FOMDA approach this represents the first level of transformation.

To each target platform combination, a distinct workflow documenting the transformation activities is needed. This motivates organizing the transformations in tiers (hence a transformation tier refers to every transformation in a target platform). Every tier receives a high-level model as input and applies some transformations, making it specific to one or more features (PIM-to-PSM transformation). The workflow, as well as the FM and the transformers, can be reused to transform any model that must use combined target platforms. Figure 5 shows an example of workflow to specify mappings and transformations based on some of selected features of Figure 3. The workflow is an activity diagram decorated with stereotypes applied to activities, objects, and transitions to reflect FOMDA meta-model elements.

Features are identified by activities decorated with the «*FeatureElement*» stereotype, and represent target platforms transformers in high-level. Those which transition contains a guard [identity], in this transformation configuration, are not able to apply transformations and then it outputs the same model received as input. This is a defined formalism to state when a feature should not apply transformations in the workflow. Nevertheless, the same feature could be used in other workflows to apply transformations.

When using FOMDA to make a M2M transformation, designer must apply mappings of a feature output to the input of some target platform from the FM. Transitions with «*FeatureOutput*» represents the feature output and with «*FeatureInput*» represents the feature input. Both transitions must be related to some object, decored with «*FeatureParameter*» stereotype. This indicates instances of FeatureParameter FOMDA meta-data and documents features' inputs and outputs.

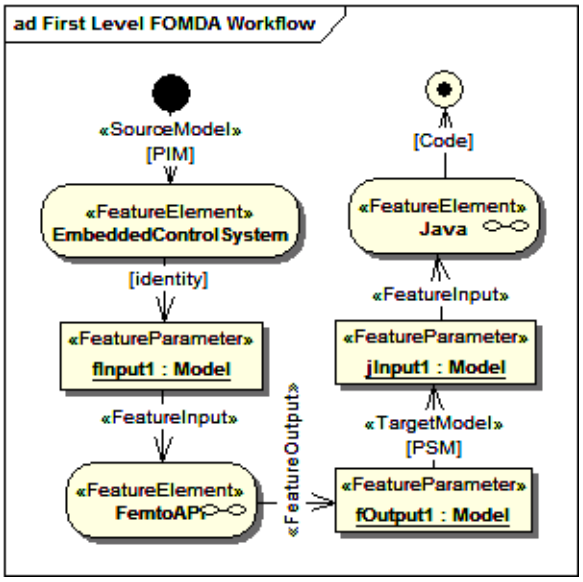


Figure 5. Example of Transformation Workflow

The output of some feature can be mapped to the input of other. The mapping of source model (this can be in the output parameter of some feature) to a target platform is documented as an action or just by transitions between objects. The transition between activities, objects and actions indicates the direction in model mapping. Mappings, in this level, represent the addition of some source model elements to some target platform. Transitions that contain the «*SourceModel*» stereotype indicate the beginning of the M2M transformation, and those with the «*TargetModel*» stereotype indicate its end.

Analyzing the workflow of Figure 5, the first step is to bring a PIM and map it to the EmbeddedControlSystem feature. This platform does not apply transformations, simply generates the identity of the PIM. The identity model (PIM) is then mapped to the feature parameter fInput1 of FemtoAPI feature just if the PIM is of mof.core.Model type. Such constraint is defined in the type of fInput1 object.

The result of feature transformation is mapped to its output parameter: the specification of transition with «*FeatureOutput*» stereotype related with object decored with «*FeatureParameter*» stereotype. The FemtoAPI output is documented in the workflow of Figure 5 by the fOutput1 object and it must be mapped to the Java feature parameter: the object jInput1 stereotyped as «*FeatureParameter*». Observe that the transition between fOutput1 and jInput1 is stereotyped as «*TargetModel*». This indicates that the target model was reached in the transformation, so M2M transformations have finished and M2C transformation may be applied. Finally, Java feature will generate java code and the transformation is complete.

In this phase we specified a workflow to guide the designer of FOMDA in mappings source model to feature parameters and to apply transformations successively while the target model is not reached. This is possible because we have configured the feature parameter types received by

features, and documented all activities that the designer must follow to apply transformations. The workflow offers views of high-level transformations, which receives source models and generate target models. The transformation itself is applied into the features, by the third and low levels transformations. Activities with sub-activity state contains another activity diagram documenting the features internal transformations.

Each sub-activity workflow shows that other actions must be realized to apply the transformation over the sub-activity state. Taking the FemtoAPI sub-activity as an example (see Figure 6), it specifies another workflow that shows the transformations and mappings executed internally to the FemtoAPI sub-activity.

The workflow in Figure 6 documents the sub-activity FemtoAPI of Figure 5. The «*SourceModel*» and «*TargetModel*» stereotypes identify the beginning and the end of M2M transformations. The designer begins by mapping the source model to the FemtoAPI input parameter. This means that designer can work with mapped value of feature parameters in the current step. They do this by taking an element from the model (that is inside the feature parameter) and this is documented as "select" action on the workflow. In the future, actions should be used by FOMDA toolkit to guide designers in mappings and transformations.

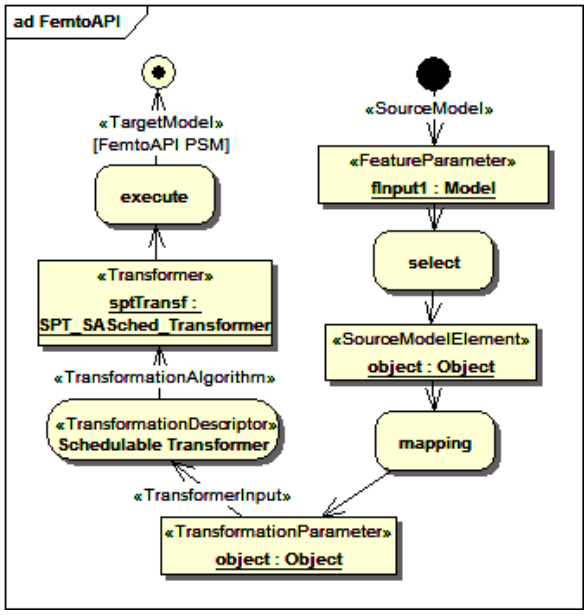


Figure 6. FemtoAPI Sub-activity Workflow

Model elements are identified as objects with «*SourceModelElement*» stereotype. Figure 6 shows the selection of a model element from the MOF type mof.collaboration.Object, which is represented in action "select" between the "fInput1" object and the "object" decorated with «*SourceModelElement*» stereotype. The selected object is then mapped (actions with "mapping" value represents the mapping of some model element) to the "object" transformation parameter, identified by object with «*TransformationParameter*» stereotypes, pursued by the Schedulable Transformer activity.

Schedulable transformer is a transformer that applies FemtoAPI target platform internal transformation and is identified by the «*TransformationDescriptor*» stereotype. The designer must then, as an action with "execute" value, execute the object indicated by the transition from Schedulable Transformer activity to the object "sptTransf". Last object is the instance of transformer that applies transformation over the "object" transformation parameter and is identified by the «*Transformer*» stereotype. The expected result is the FemtoAPI PSM indicated in the transition from the "sptTransf" to end state. The FemtoAPI sub-activity has finishes. But the designer must follow the transition from it to the next activity or object identified in the workflow of Figure 5.

Transitions stereotyped with «*TransformationAlgorithm*» stereotype, configures the transformation algorithm into the transformation descriptor, and with «*TransformerInput*» stereotype configures the transformation parameters of some transformer.

3.5 Toolkit Prototype

The FOMDA toolkit prototype was developed to test the proposed extensions in the FM to apply transformations. Using the toolkit designer can specify FMs and make the instantiation process highlighted in section 2.2. Afterwards, the designer can start the configuration of features to apply transformations, beginning from features parameters and adding transformers to the features. He can make mappings between features without configuring features input parameters and, in this case, any feature must receive a mof.core.Model parameter type. Therefore, mappings between features can be done without restrictions.

Designer can map elements from the source model received by some feature to the required transformation parameters of each transformer contained by that feature. To do this, the application designer follows the mappings and transformations specified in the workflow.

4 Case Study

This section presents a case study that exemplifies the generation of an ERTS using the FOMDA approach. The system under consideration is the movement controller from a motorized wheelchair, which is detailed in [12]. This system is designed in UML in a platform-independent manner and contains real-time restrictions coming from the UML Profile for Schedulability, Performance, and Time (SPT). Figure 7 presents two collaborating objects (extracted from a more general diagram) that are used to exemplify the proposed mappings.

The objects on Figure 7 were selected for this study because they contain important timing information on which respective implementation is highly platform-dependant. The following aspects, for instance, should be observed during the implementation: time management, concurrent and periodic execution, and also deadline control. Therefore we are challenged by the following problem: how to generate the final ERTS according to the platform selection of the application designer? This section guides the reader through the FOMDA approach to solve the referred problem.

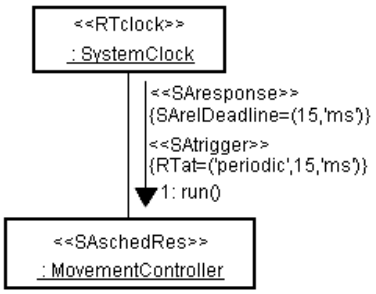


Figure 7. Wheelchair Movement Control System

4.1 Configuring the Target Platform

Before thinking in model transformations one should consider the possible configurations of the target platform. This configuration is performed by creating the FM as exemplified in Figure 2. Moreover it is also required to design the transformers for each possible configuration, together with the expected behavior for each transformer, i.e. the required input and expected output. Such behavior is described in the so called FOMDA workflow. Considering as example the selected features from Figure 3, a workflow with about 12 mappings and transformations is required, one for each selected feature.

Mappings and transformations must be represented in high-level, using a workflow to show the order that they are applied. This is done by specifying activities, related to the selected features from FM, objects, representing feature inputs and outputs, and transitions between source and target platforms. The workflow shown in Figure 5 must be readapted to contain plus 9 mappings and transformations, documenting all selected features of FM shown in Figure 3, as explained in Section 3.6.

Some of the high-level transformers presented in Figure 5 are further sub-divided, such as FemtoAPI and Java. Taking the FemtoAPI transformer as example it results in five other specifications, which are shown in Figure 6. The workflow from Figure 6 describes the FemtoAPI low-level internal mappings and transformations and was explained in Section 3.6. The object decorated with «*Transformer*» stereotype is a reference to the SPT_SAsched_Transformer class that contains the transformation algorithm used by FemtoAPI feature to transform a model. This algorithm is presented in Figure 8.

To demonstrate the transformation of the source model from Figure 7, a M2M transformation is applied using the transformer of Figure 8. The transformer must receive as a parameter an element of type mof.core.Object and, based on the SPT profile, should apply a transformation from source-model to the FemtoAPI target model. Analyzing the transformer code, in line 5 there is the code to get the transformation parameter "object". Thus designer must map as transformation parameter an element from source model that is instance of mof.collaboration.Object. Line 7 shows how to take the target class from that object, based on the MOF specification. The target class is the element of source model that we apply M2M transformation, making it dependent from FemtoAPI target platform. Line 10 shows how a FOMDA transformer tests if some feature from the FM is selected. When the feature "Concurrent" is selected in the FM or the target class is active or it contains the «*SAschedRes*» stereotype, then the target class must inherit from "RealtimeThread" class. Such mapping is based in the previous works from Becker et al in [12][6].

```
1  public class SPT_SASched_Transformer extends
AbstractTransformer {
2   protected mof.core.Class targetClass;
3   protected mof.collaboration.Object obj;
4   public Object doTransformation() throws Exception {
5    TransformationParameter tp1 =
observer.getParameter("object");
6    this.obj = (mof.collaboration.Object) tp1.getElement();
7    this.targetClass = (mof.core.Class) obj.getType();
8    if (targetClass == null) throw new
       Exception("Parameter object must contains a type");
9    Stereotype saschedulable =
obj.getAssignedStereotype("SAschedRes");
10   if (isFeatureSelected("Concurrent") ||
         targetClass.isActive() || targetClass.
          containsStereotype("SAschedRes")){
  /*the target class must inherit from RealtimeThread class*/
11     Class superClass = new
          Class(targetClass.getRootParent());
12    superClass.setName("RealtimeThread");
13    targetClass.getRootParent().addElement(superClass);
14    targetClass.getSuperClasses().add(superClass);}
15    searchForResponseStereotype();
16    return null;}
17  private void searchForResponseStereotype() {
18         …
19     createMainTaskOperation();
20     createExceptionTaskOperation();
      }
21  private void searchForSatriggerStereotype(){
22
    }
23  private void createMainTaskOperation() {
24     Operation op = new Operation();
25     op.setName("mainTask");
26     op.setVisibility("public");
27     op.setScope("instance");
28     op.setAbstract(false);
29     Parameter ret = new Parameter(op);
30     ret.setName("Return_"+op.getName());
31     ret.setKind("return");
32     DataType voidDT = new DataType(op.getRootParent());
33     voidDT.setName("void");
34     op.getRootParent().addElement(voidDT);
35     ret.setType(voidDT);
36     op.addElement(ret);
37     Comment mTaskCode = new Comment(op.getRootParent());
38     mTaskCode.addAnnotatedElement(op);
39     String strCode = "";
40     …/* designer must write algorithm to define the code
                of  the"mainTask" operation */
41      mTaskCode.setBody(strCode);
42     op.addComment(mTaskCode);
43     targetClass.addElement(op);
      }
  }
```

Figure 8. Schedulable FemtoAPI Target Platform Transformer

From lines 18 to 22 the designer must write an algorithm to apply transformation over messages decorated with the «*SAresponse*» stereotype. Messages with such stereotype are mapped to FemtoAPI target platform by adding two operations to target class: "mainTask" and "exceptionTask". Lines 23-43 contain the algorithm to add the "mainTask" operation to the target class, where the operation body (it determines the implementation code for the operation) would be generated in the code from lines 39-41. Such code would be used to add FemtoAPI java code to the associated element, and can still be used afterwards by some other transformer. In this step, we are interested in applying M2M transformation to modify a class and not to generate its code.

The target classes, for instance, are not ready for code

generation because the mainTask operation, as well as the other target class elements, must contain code sections inside the elements body, as done in [6][12]. These elements body code sections, like for mainTask operation, can come from the transformation of many of other source model constraints and each of code section must be organized to generate the correct code. Our solution to generate many of elements body code sections is to add such sections as a comments associated with the elements. Afterwards such comments can be arranged into elements body and a M2C transformer can then generate the correct code.

Lines 21-22 the designer can write the algorithm to search for messages decorated with «*SAtrigger*» stereotype. The algorithm must add a "mof.core.Attribute" instance type to the target class. The {RTat = ('periodic',15,'ms')} tagged value from «*SAtrigger*» stereotype in Figure 7 indicates that several attributes containing FemtoAPI specific data types must be generated (e.g. PeriodicParameters, AbsoluteTime, etc) and initialized, also following the mapping defined by Becker et al in [6][12].

4.2 Features Selection and System Generation

Every time a new application is designed the application designer must perform the FM instantiation and use the workflow to apply transformations into the source model, since it documents mappings and transformations using selected features. Nevertheless, this is not a problem because application designer must apply successive mappings and transformations guided by several workflows. As more target platforms are modeled, more M2M transformations becomes necessary. FOMDA toolkit helps application designer in configuring target platforms using FM to apply such transformations and to compose transformation using the third-level transformation, all using FOMDA workflow definition. Once such resources are configured, designer can use it to generate code to the target platforms selected in the FM.

4.3 Changing the Target Platform

In this case study we presented an example of using FOMDA to configure transformations for a specific features combination, related to the FM presented in Figure 3, and to use it to transform any model that needs such configuration. Naturally, as more platforms become necessary to a system, new configurations have to be established. When all combinations are fulfilled, then designer has a powerful resource to generate code to many target platforms.

As an example in target platform changes, if a new system also requires a middleware such as Embedded CORBA, then "Middleware" and "EmbCORBA" features must be selected in the FM. This selection defines a new configuration and a new FOMDA workflow must be specified. The feature "EmbCORBA" could be used to transform the output of "FemtoAPI" activity and configure it to work with embedded CORBA target platform. This transformation could be of M2M category. Further, the output of "EmbCORBA" could be mapped to "Java" activity and then a new code could be generated to supply both embedded CORBA and FemtoAPI target platforms.

When system non-functional requirements changes, specially the architectural ones, a re-adaptation of FOMDA workflow becomes necessary. Transformers, features parameters, etc, can be reused without changes even if target platform changes. The low-level transformers can be reused even in change of the target platforms or in the order that they are applied. Changes in transformation order for some target platform affect the high-level FOMDA workflow, but it is easy to re-adapt it for the new requirements. In the future, the FOMDA toolkit should be extended to help in workflow specification and to use it as a wizard to guide application designer in the mappings and transformations of source models to target platforms.

5 Related Works

Almeida et al. in [7] proposed a manner to apply consecutives model-to-model transformations. They do it by defining transformations tiers as abstract platforms that are represented as extensions of MOF packages. Nevertheless, it is not possible to have the notion about the order in which transformations occurs. Tekinerdogan et al. in [45] have suggested the usage of FM in mappings and transformations of models. Almeida and Tekinerdogan proposals are very interesting as they identify target platforms and use it to map and transform models. FOMDA approach has a complementary contribution, proposing to organize transformation in tiers, with every tier representing a target platform. Moreover FOMDA innovates by using FM in conjunction with MDA.

Other tools addressing direct model manipulation category are UMT [15][19] and Jamda [6]. FOMDA toolkit also works with direct model manipulation but only in one from its four levels of transformations. Application designer can make mappings and transformation over target platforms in a high-level tier and use, inside platforms, transformers to apply a transformation direct to some model element in a low-level tier.

6 Conclusions

This paper described the problem of ever changing embedded systems non-functional requirements, specially architectural ones, and proposed a solution based on Features Model and in MDA standards. We proposed the Features-Oriented Model-Driven Architecture (FOMDA) approach to help application designers in defining mappings and transformations of any model to as many target platforms as wished, by configuring model-to-model and model-to-code transformations over tiers. Every tier represents some target platform that the system must be mapped and transformed to. We divided the task of configuring mappings and transformations in four levels to simplify and clarify the transformations.

Our low-level transformation definition is characterized as one of direct manipulation approach, and designer must write transformers algorithms to apply transformations over source model elements in conjunction with third level transformation organization. The second-level transformation is to configure inputs and transformers of target platforms. Finally the first-level transformation represents a workflow to guide application designer with mappings and

transformations over target platforms.

A case study was presented to evaluate the application of FOMDA approach in the development of an embedded real-time system. It presented how to make a general high-level UML model (PIM) specific to a target platform (PSM). Obtained results are optimistic and lead to the conclusion that FOMDA approach can make designers re-think their current development process to make it more decoupled from a specific target platform.

As future work authors intend to extend the current version of the FOMDA toolkit to help directly in the workflow specification and to use it as a wizard to guide application designer in making mappings and transformations from source models to target platforms.

7 Acknowledgments

Thanks are given to the Brazilian funding agencies CNPq and CAPES for supporting this project.

8 References

[1] B. Selic. “On Software Platforms, Their Modeling with UML 2, and Platform-Independent Design,”. In Proc of .8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. Seattle, USA, 2005, pp. 15-21

[2] B. Tekinerdogan, S. Bilir, and C. Abatlevi. Integrating Platform Selection Rules in the Model Driven Architecture Approach. In Proc. of Model-Driven Architecture: Fundations and Applications, June 2004. pp 184-200.

[3] C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse method with domain-specific architectures. Annals of Software Engineering, V5, Balzer Science Publishers, 1998, pp. 143-168.

[4] D. Garlan, and M. Shawn. Software Architecture: Prespectives on an Emerging Discipline, Prentice Hall, April 1996.

[5] G. Boch, J. Rumbaugh, and I Jacobson, The Unified Modeling Language: User Guide, Addison-Wesley-Longman, 1999.

[6] Jamda: The Java Model Driven Architecture. Version 0.2.4, November 2003. Available at <http://sourceforge.net/projects/jamda/>.

[7] J. Almeida, R. Dijkman, M. Sinderen, and L. Pires. Platform-independent modeling in MDA: Supporting abstract platforms; In Proc. of Model-Driven Architecture: Fundations and Applications, June 2004. pp 217-231.

[8] K. Czarnecki K., H. Simon. Classification of Model Transformation Approaches. In Proc. of OOPSLA’03 Workshop on Generative Techniques in the Context of Model-Driven Architecture; 2003

[9] L. Becker, R. Höltz, C. Pereira. On Mapping RT-UML Specifications to RT-Java API: Bringing the Gap; In Proc. of ISORC 2002, Crystal City, USA. pp. 348-355.

[10] M. Fontoura. A systematic Approach to Framework Development. Ph.D. Thesis, Computer Science Department, Pontificial Catholic University of Rio de Janeiro (PUC-Rio), 1999.

[11] M. L. Griss. Product-Line Architectures, Chapter 22. Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley, 2001.

[12] M. Wehrmeister, L. Becker, and C. Pereira. Applying the SEEP Method in the Design of a Real-Time Embedded Control System for a Motorized Wheelchair. In proc. of 10th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA), Catania, It, 2005.

[13] Object Management Group; Meta Object Facility (MOF) 2.0 Core Specification. October 2004. Available at <http://www.omg.org/cgi-bin/apps /doc?ptc/03-10-04.pdf>.

[14] Object Management Group MDA Specifications. October 2004. Available at <http://www.omg.org/ /mda/specs.htm>.

[15] QVT-Partners; Revised submission for MOF 2.0 Query / Views / Transformations RFP; Version 1.1. October 2003. Available at <http://qvtp.org/>.

[16] S. Deelstra, M. Sinnema, J. Gurp, and J. Bosch. Model Driven Architecture as Approach to Manage Variability in Software Product Families. Workshop on Model Driven Architecture: Fundations and Applications, June 2003. pp 109-114.

[17] T. Oliveira, I. Filho, C. Lucena, P. Alencar, and D. Cowan. Enabling Model Driven Product Line Architectures. Second European Workshop on Model Driven Architecture (MDA). Canterbury, England. September 2004,

[18] U. Eisenecker, and K. Czarnecki. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[19] UMT-QVT. June 2005. Available at <<http://umt-qvt.sourceforge.net>>