Uma Solução para Reuso e Manutenção de Transformadores de Modelos Usando a Abordagem FOMDA

Fabio Paulo Basso[1], Leandro Buss Becker[2], Toacy Cavalcante Oliveira[1]
[1]Instituto de Informática – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

[2]Departamento de Automação e Sistemas – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88.040-900 – Florianópolis – SC – Brasil

fabiopbasso@gmail.com, lbecker@das.ufsc.br, toacy@inf.pucrs.br

Resumo. *A principal motivação para uma corporação usar a MDA é ganhar em produtividade, permitindo aos projetistas migrarem modelos de sistemas mais genéricos para domínios específicos. Atualmente, um grande desafio ao utilizar a MDA é o de efetuar a manutenção dos transformadores de modelos em decorrência da modificação das características utilizadas para desenvolver sistemas. A manutenção dos transformadores de modelos é uma tarefa importante na MDA e deve ser discutida em detalhes. Este documento apresenta a abordagem Features-Oriented Model-Driven-Architecture (FOMDA) como uma solução para garantir a manutenibilidade de transformadores de modelos. Esta solução é detalhada em um estudo de caso de desenvolvimento de interfaces gráficas de usuário (GUIs) em um ambiente corporativo.*

Abstract. *The main goal for a company to use MDA is to gain in productivity, allowing designers to quickly migrate more generic system models to specific domains. Currently, a great challenge when using the MDA is related to the model transformers maintenance according to the modification of the characteristics used to develop systems. The maintenance of the models transformers is an important task in the MDA and should be discussed in details. This paper presents the Features-Oriented Model-Driven-Architecture (FOMDA) approach as a solution to model transformers maintance. Such solution is detailed in an enterprise environment case study of a graphic user interface development (GUIs).*

1. Introdução

Muitas abordagens para a Model Driven Architecture (MDA) [5][7][8][11][14][19][20] estão sendo utilizadas na produção de software e possibilitam aplicar transformações de modelos UML[13] para determinadas características usadas no desenvolvimento de sistemas [4][10]. Entretanto, o reparo e o reuso dos transformadores de modelos, que são tarefas necessárias em decorrência de modificações na configuração dessas características, ainda são tarefas complexas de serem realizadas [3][10][18]. Em muitos casos estas modificações implicam no descarte dos transformadores de modelos. Para resolver este problema este documento apresenta a abordagem FOMDA que possibilita que transformadores possam ser compostos, reutilizados e alterados dinamicamente de acordo com a mudança das características usadas no desenvolvimento de sistemas.

FOMDA é uma abordagem que foca no reuso e manutenção de transformadores em decorrência da troca de plataformas-alvo. Esta abordagem define transformadores de

modelos de alto e baixo nível e separa a especificação de transformações em quatro etapas: Especificações de *workflows* são utilizadas para organizar e documentar transformações de modelos; a especificação de um *Modelo de Features* [5][6][22] é utilizada para organizar características de sistemas; também a composição dinâmica de transformadores e a especificação dos algoritmos de transformadores de modelos utilizando soluções existentes para transformação na MDA.

Este documento apresenta a FOMDA em detalhe, demonstrando-a em um estudo de caso. Este estudo de caso foi realizado em um ambiente corporativo e descreve como a abordagem FOMDA supera o desafio de reuso e manutenção de transformadores de modelos em decorrência de mudanças nas características utilizadas para desenvolver aplicações. Os resultados obtidos são satisfatórios e indicam que esta abordagem é muito útil para o reuso e a manutenção dos transformadores de modelos.

O restante do trabalho está organizado como segue. A Seção 2 descreve a abordagem FOMDA. A Seção 3 apresenta o estudo de caso e exemplifica o reuso e a manutenção de transformadores. Os trabalhos relacionados são apresentados na Seção 4. As contribuições, restrições e os trabalhos futuros são apresentados na Seção 5.

2. Abordagem FOMDA

A abordagem FOMDA propõe a organização de transformações em quatro etapas (ver Figura 1). Estas etapas podem ser divididas em transformações horizontais (TH) e verticais (TV) representadas por setas nesta figura. Os artefatos definidos nessa figura permitem a um projetista de aplicações executar as ações relacionadas com cada etapa dessa abordagem de acordo com determinados eventos. As transformações verticais *TV1'*, *TV2'* e *TV3'* utilizam os artefatos gerados pelas transformações *TV1*, *TV2* e *TV3*.

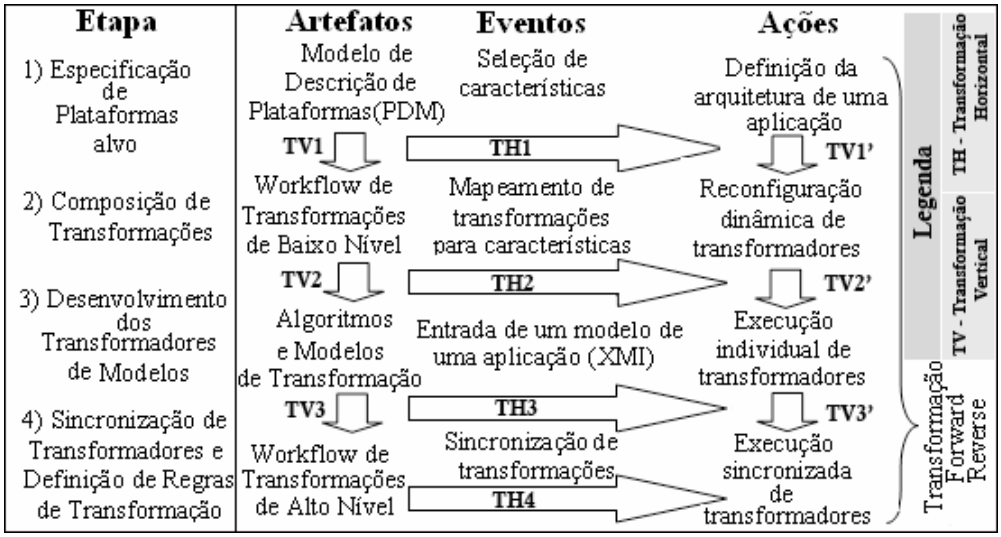

Figura 1. Abordagem FOMDA

A abordagem FOMDA inicia pela transformação horizontal denominada *TH1*. Esta transformação utiliza um Modelo de Descrição de Plataformas (PDM) e gera como resultado a definição de uma arquitetura de aplicação. O PDM possibilita realizar a primeira transformação vertical *TV1* que organiza as transformações de modelos e as características de plataformas em um *workflow*. Um *workflow* nesta abordagem é um transformador de modelos e define entradas (modelos fonte) e saídas (modelos alvo). A transformação vertical *TV1* é realizada para avaliar a reutilização de transformadores.

Esta transformação divide a transformação de modelos em transformadores mais especializados para determinadas características do PDM. Nesta etapa a transformação *TV1'* permite executar transformações com base na seleção das características da arquitetura selecionada na transformação horizontal *TH1* e organizada em *TH2*.

Na segunda etapa da abordagem FOMDA a transformação *TH2* pode ser realizada. Esta transformação tem por objetivo modificar o *workflow* de transformações de baixo nível para possibilitar a composição de novas transformações dinamicamente no caso de mudança na seleção das características realizadas na transformação *TH1*. É essa transformação que determina se os transformadores de modelos são reutilizáveis no caso de troca das plataformas alvo.

Na terceira etapa de transformação de modelos um modelo de entrada especificado em um documento no formato XMI [18] deve ser transformado para as características selecionadas no PDM. Isto é realizado de acordo com o *workflow* resultante da transformação *TV1*. Para realizar a transformação é necessário ligar as atividades de transformação definidas no *workflow* com algoritmos de transformação de modelos. Os algoritmos são definidos na transformação vertical *TV2*. A transformação *TH3* é caracterizada por transformar modelos fonte para modelos alvo que são especificados em atividades de transformação do *workflow*. Neste estágio as características do PDM estão organizadas em atividades de transformação de modelos no *workflow* e estas atividades estão ligadas com algoritmos de transformação. Então, a transformação *TV2'* pode ser executada, permitindo que a ação execuções individuais de transformadores seja realizada.

A quarta etapa de transformação de modelos sincroniza as saídas e entradas dos *workflows* de baixo nível. Os *workflows* de baixo nível podem ser combinados em um *workflow* de alto nível, compondo assim outros transformadores de modelos. Isto é realizado na transformação *TV3*. Alguns *workflows* de baixo nível podem gerar como resultado outros modelos, em transformações conhecidas como model-to-model (M2M) [16]. Os modelos resultantes podem ser usados como entradas para outros *workflows* de baixo nível. Isto é denominado de sincronização de modelos que é realizado na transformação horizontal *TH4*. A transformação *TV3'* determina então a execução de transformações M2M sincronizadas pelo *workflow* de alto nível.

A abordagem FOMDA permite executar transformações *forward* e *reverse* [16] com base nas quatro etapas de transformação de modelos. Estas transformações podem ser M2M ou model-to-code (M2C). As próximas seções detalham cada uma destas etapas.

2.1 Especificação de Plataformas Alvo

O exemplo do Modelo de Features (FM) mostrado na Figura 2 especifica possíveis requisitos não funcionais para o desenvolvimento de Interfaces Gráficas de Usuário (GUI) utilizando o formalismo descrito por Czarnecki et. al em [22]. Neste modelo o projetista precisa identificar requisitos como de linguagens de programação, bibliotecas, *frameworks*, componentes de software e etc. para utilizá-los para aplicar transformações de modelos fontes em modelos alvo desses requisitos.

O modelo especificado na Figura 2 é um PDM que define que qualquer GUI deve ter pelo as seguintes características: interface gráfica para *Desktop* (usando ou *Swing* ou *Awt*) e/ou para *Web* (usando *Jasper* e/ou *JSP* e/ou *JSTL*), um mecanismo conhecido como *binder* para inserir os dados de um objeto nos campos de um formulário e vice-versa (que deve ser realizado ou com *Commons Binder* ou manualmente com *Manual Bind*); um mecanismo para adaptar internacionalização na interface gráfica, que pode

ser ou utilizando *I18N* que pertence à linguagem *java* ou o *Res Msg* que é um componente de uso próprio. A característica de validação deve ser opcional, porém quando ela estiver selecionada, então se deve escolher pela validação manual (característica *Manual Val*) ou por utilizar a característica *Commons Validator*. No caso de usar *Commons Validator*, então obrigatoriamente a característica *Commons Binder* deve ser selecionada.

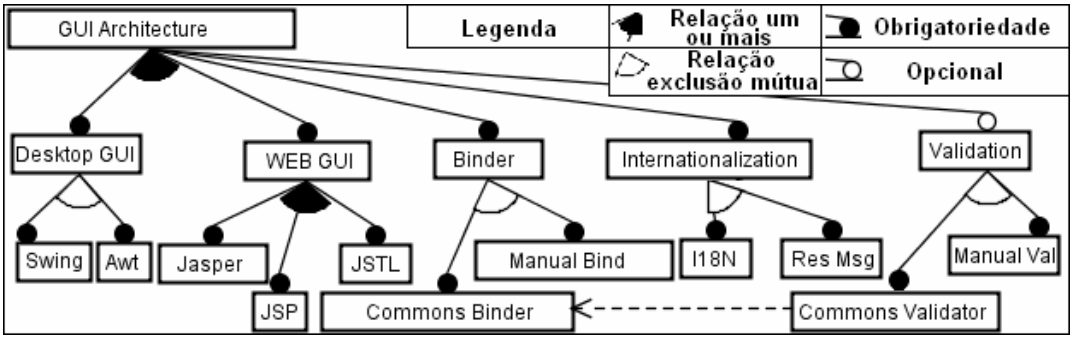

Figura 2. Exemplo de um Modelo de Features com Características de Arquiteturas

Após a realização da primeira transformação horizontal *TH1*, o projetista pode realizar uma transformação vertical *TV1* a fim de mapear transformações para as características definidas no PDM. O FM descrito na Figura 2 pode auxiliar um projetista na execução dessa transformação vertical. Por exemplo, o modelo fonte GUI pode ser mapeado e transformado para a característica *Swing*. Este mapeamento só é possível se características de *Binder* e *Internacionalization* também forem mapeadas e transformadas, dado a sintaxe especificada no FM. É possível perceber que, de acordo com a seleção destas características, podem existir combinações que representam o modelo GUI transformado para a característica *Swing*. Esta variabilidade sugere o uso de decomposição das transformações.

A abordagem FOMDA beneficia-se do aspecto variável da seleção do FM para garantir o reuso de transformações de modelos. A próxima atividade dessa abordagem é a decomposição de uma transformação de modelos em transformações especializadas para as características definidas no PDM. Esta atividade é realizada em uma transformação horizontal *TH2* que visa organizar as características do FM e os transformadores de modelos ditos de baixo nível. Para isso a abordagem FOMDA descreve uma etapa denominada composição dinâmica de transformações.

2.2 Composição de Transformações

A composição de transformações é a segunda etapa da abordagem FOMDA. Nesta etapa os transformadores de modelos podem ser combinados dinamicamente. Para compor transformadores esta abordagem propõe um projeto de transformadores. Este projeto é feito utilizando a UML 2 e estende o padrão Query, View and Transformations (QVT) [18]. A próxima Seção descreve como desenvolver um projeto de transformadores.

2.2.1) Projeto de Transformadores

Um projeto de um transformador é caracterizado como um *workflow* de baixo nível e é definido na transformação vertical *TV1*. Ele é um Diagrama de Atividades que contém atividades, objetos, ações, transições e comentários decorados com estereótipos

definidos na abordagem FOMDA. Estes estereótipos visam especificar transformadores de modelos e ações tomadas por transformadores durante a execução de transformações. A Figura 3 exemplifica um *workflow* em que o estado inicial determina o tipo de modelo fonte utilizado como entrada de um transformador de modelos. Isso é feito usando uma transição decorada com o estereótipo «*SourceModel*» do estado inicial para um dos parâmetros de entrada desse transformador. Parâmetros de entrada são especificados marcando objetos com o estereótipo «*TransformationParameter*» e são ligados à transformadores por transições decoradas com o estereótipo «*Parameter*». Parâmetros de entrada determinam os requisitos para a execução da transformação realizada pelos elementos do *workflow*.

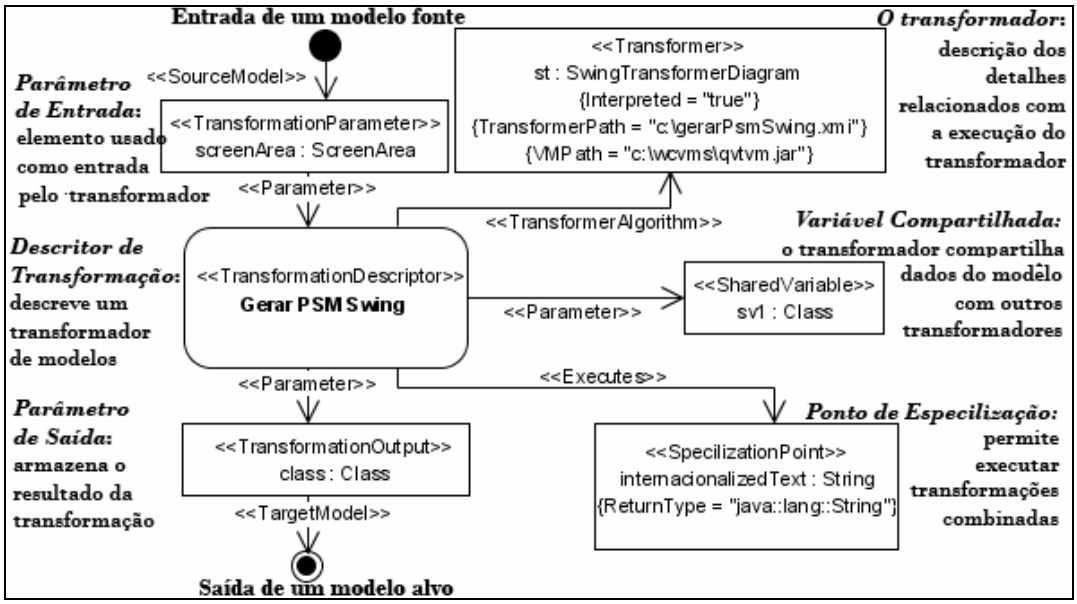

Figura 3. Composição de Transformadores de Modelos

Transformadores são especificados em atividades decoradas com o estereótipo «*TransformationDescriptor*». Eles são chamados de atividades de transformação e contêm informações sobre os algoritmos de transformação de modelos dos transformadores. A execução de uma atividade de transformação implica na execução do transformador que está ligado a ela por uma transição decorada com o estereótipo «*TransformerAlgorithm*». Esta transição deve ter como alvo um objeto decorado com estereótipo «*Transformer*», que descreve os dados sobre a execução do transformador de modelos. Quando executada uma atividade de transformação, os parâmetros de entrada dela são adicionados nos parâmetros do transformador que é, então, executado. O resultado dessa execução é um dado que é armazenado em um parâmetro de saída da atividade de transformação (objeto decorado com o estereótipo «*TransformationOutput*»). Parâmetros de saída descrevem o tipo dos modelos alvo que se espera como resultado da transformação realizada no *workflow*.

O *workflow* de baixo nível descrito na Figura 3 especifica a transformação de um modelo fonte (objeto de nome *screenArea* que é uma instância de *GUYLayout::ScreenArea* descrito no *GUI Profile* [15]) em um modelo alvo (objeto de nome *class* que é uma instância de *UML::Classes::Kernel::Class*). O transformador que realiza esta transformação é a atividade de transformação denominada "*Gerar PSM Swing*". É possível visualizar no *workflow* que este transformador é o executor da

transformação, porque existe uma transição (decorada com o estereótipo «*SourceModel*») do estado inicial para um objeto que especifica um parâmetro de entrada deste transformador. Atividades de transformação podem definir ainda variáveis compartilhadas por transformadores e pontos de especialização que são parâmetros de transformadores explicados mais adiante. A transformação de um elemento GUI é finalizada quando o estado terminal do *workflow* for alcançado na transformação e resulta em um modelo alvo identificado pela transição decorada com o estereótipo «*TargetModel*». A Tabela 1 descreve alguns dos elementos utilizados nos *workflows* da abordagem FOMDA que não foram explicados neste documento.

Tabela 1 - Tags Utilizadas para Especificar Detalhes de Transformadores

{ExecutionKind} É aplicada em Transições
Especifica o tipo de execução da transformação: o padrão é (*forward*) e indica a execução de uma transformação de um modelo em alto nível para um modelo de baixo nível; (*reverse*) indica a execução de uma transformação reversa, ou seja, de um modelo em baixo nível para outro de alto nível.
{TransformationKind} É aplicada em Atividades
Pode ser utilizada em atividades de transformações decoradas com o estereótipo Transformation Descriptor. Descreve qual é o tipo de transformação utilizada podendo conter os seguintes valores: se transformação de manipulação direta (*TMD*) é o *default*; se baseada em grafos (*Graph*); se baseada em engenharia de domínio (*DE*); se interpretada (*Virtual*).
{Interpreted} É aplicada em Objetos
Especifica se o transformador deve ser interpretado por um programa ou máquina virtual de uma ferramenta MDA. Aceita valores (*true*,*false*), sendo *false* o valor *default*.
{TransformerPath} É aplicada em Objetos
Especifica o caminho do arquivo (*jar*, *xmi*, *xml*, etc) que define o algoritmo do transformador.
{VMPath} É aplicada em Objetos
Especifica o caminho de um interpretador que interpreta o algoritmo do transformador e efetua as transformações. Este interpretador é uma máquina virtual desenvolvida especialmente para cada ferramenta de MDA e cada abordagem ou linguagem de transformação de modelos.

2.2.2) Composição Dinâmica de Transformações

Até o momento foi exemplificado como especificar um projeto simples de transformadores usando o *workflow* de baixo nível. No entanto, o FM mostrado na Figura 2 sugere que as transformações sejam combinadas de acordo com as características selecionadas nesse modelo. Neste sentido, o transformador para a característica *Swing* deve ser especializado para uma das características de internacionalização, devido à sintaxe estabelecida entre as características do FM. Para isso, é necessário indicar qual o transformador é executado para efetuar transformações para as características selecionadas: se o transformador para a característica *I18N* ou para *Res Msg*. O uso de pontos de especialização nos *workflows* de baixo nível possibilita que transformadores sejam executados de acordo com a seleção das características do FM. Isto é realizado na transformação *TH2*.

Um ponto de especialização é um parâmetro de um transformador e também é um transformador. Este parâmetro pode ser executado explicitamente por este transformador em um trecho de seu algoritmo de transformação. Isto significa dizer que na codificação de um transformador do tipo de manipulação direta de modelos (TMD) [16] é feita uma chamada para a execução dos seus pontos de especialização. No *workflow* da Figura 3 o transformador "*Gerar PSM Swing*" possui um ponto de especialização identificado por um objeto de nome "*internationalizedText*" e está decorado com o estereótipo «*SpecializationPoint*». Este ponto de especialização é um parâmetro que recebe do transformador "*Gerar PSM Swing*" objetos do tipo string e

efetua uma transformação. Além disso, é necessário informar nos pontos de especialização o tipo de retorno de sua transformação. A *tag {ReturnType=tipo de dado}* é utilizada com este propósito.

A transformação realizada por um ponto de especialização não é codificada em um algoritmo como é em transformadores TMD. Um ponto de especialização executa transformadores para obter o resultado que é esperado pelo transformador que o executou. O resultado retornado pelo ponto de especialização é uma instância do tipo de retorno especificado na *tag ReturnType*. No caso do ponto de especialização especificado na Figura 3, o resultado esperado pelo transformador "*Gerar PSM Swing*" é uma string e transformadores associados com esse ponto precisam retornar objetos deste tipo. Isto e detalhado na Figura 4.

Pontos de especialização podem conter ações. Estas ações são decoradas com o estereótipo «*Query*» e avaliam expressões em elementos do modelo. As expressões são os nomes das ações. Elementos avaliados por uma expressão são alvos de uma transição decorada com o estereótipo «*TargetElement*». Quando muitos elementos estiverem relacionados com uma ação, então por *default* a expressão tem o comportamento de uma cláusula AND. Ações podem ser aninhadas e permitem o uso de outras cláusulas, bastando informá-las como expressões. A ação localizada no canto inferior da Figura 4, mais ao centro, avalia se a característica *I18N* está selecionada no modelo. Caso ela esteja selecionada, então o transformador "*Gerar Código I18N*" é executado pela ação. Chamadas de execuções são especificadas no *workflow* com transições decoradas com o estereótipo «*Executes*».

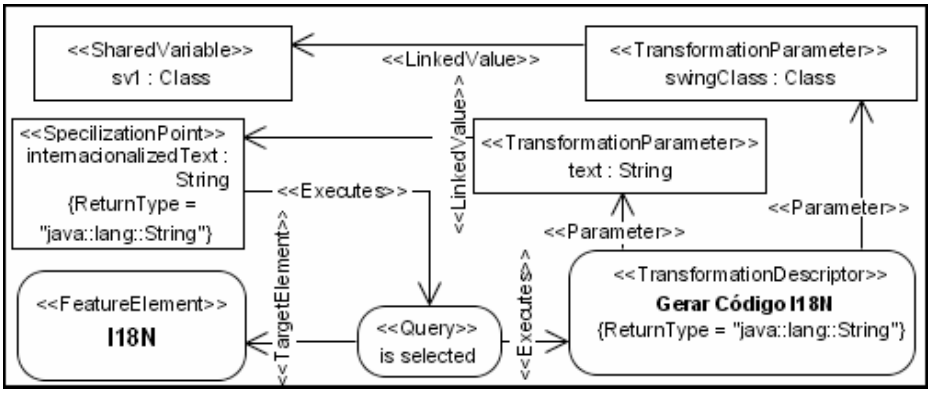

Figura 4. Recursos para Composição Dinâmica de Transformações

Transformadores somente podem ser executados por um ponto de especialização quando estes transformadores tiverem um parâmetro de transformação o qual o tipo de dado do objeto que define este parâmetro no *workflow* é o mesmo que o tipo do objeto que define o ponto de especialização. No *workflow* das figuras 3 e 4 o ponto de especialização "*internationalizedText*" e o parâmetro "*text*" coincidem no tipo de dado dos objetos que os representam. Além do mais, o tipo de retorno dos transformadores deve ser o mesmo tipo de retorno do ponto de especialização. No *workflow* da Figura 4 o ponto de especialização e o transformador "*Gerar Código I18N*" estão coincidindo nestas informações, validando assim a combinação.

Um transformador também pode conter variáveis compartilhadas. Uma variável compartilhada é um parâmetro que armazena um valor que pode ser utilizado por outros transformadores. O uso de variáveis compartilhadas permite que transformadores disponibilizem elementos de um modelo para outros transformadores. Sua representação

no *workflow* é um objeto decorado com o estereótipo «*SharedVariable*». Parâmetros de transformação podem conter dependências por variáveis compartilhadas e por pontos de especialização. Estas dependências devem ser transições estereotipadas como «*LinkedValue*» e determinam que o valor do parâmetro de transformação fonte da transição é o valor armazenado pelo parâmetro que é o alvo desta última. A combinação entre variáveis compartilhadas, parâmetros de transformação e pontos de especialização é muito vantajosa para projetar transformadores, visto que não há a necessidade de parametrizar operações no código dos transformadores como é realizado em abordagens baseadas no padrão QVT [19].

Depois de identificar a composição de transformadores é necessário desenvolvê-los. Estes transformadores podem ser de quaisquer tipos, como dos tipos TMD [2][10][16], Transformação dirigida por Estruturas ou Grafos (TEG) [8][12][19], transformação com base em Arquiteturas para Linhas de Produto (PLA) [5][6][9][11][20][21][22] dentre outras. O desenvolvimento dos transformadores é definido na terceira etapa de mapeamento e transformação de modelos.

2.3 Desenvolvimento de Transformadores

O *workflow* de baixo nível descrito nas figuras 3 e 4 permite analisar as transformações que devem ser realizadas por cada transformador individualmente. Na abordagem FOMDA o transformador *default* é do tipo TMD e precisa ser codificado em uma linguagem para transformação de modelos na transformação vertical *TV2*. A exemplificação de desenvolvimento de transformadores TMD é realizada no estudo de caso. A Figura 5 exemplifica um transformador de modelos utilizando o padrão QVT.

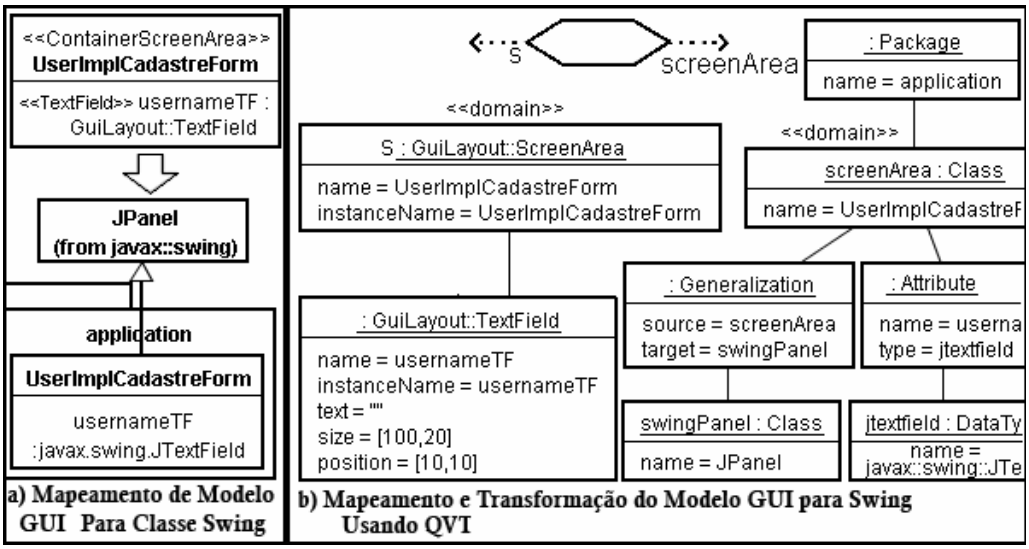

Figura 5. Exemplo de Transformação de Baixo Nível Utilizando QVT

Para associar um algoritmo de transformação a uma atividade de transformação é preciso utilizar uma transição decorada com o estereótipo «*TransformerAlgorithm*» para um objeto decorado com o estereótipo «*Transformer*». Esta transição deve ter como fonte uma atividade de transformação decorada com o estereótipo «*Transformation Descriptor*», como é exemplificado na Figura 3. As *tags Interpreted*, *TransformerPath* e *VMPath*, descritas na Tabela 1, podem ser utilizadas para especificar mais detalhes sobre o transformador ligado à atividade de transformação. Estas *tags* permitem a

definição de detalhes vinculados com a execução de transformadores não nativos da ferramenta MDA em que o transformador deve ser executado. Isto significa que os transformadores podem ser especificados em diversas linguagens e abordagens de transformação de modelos.

O objeto decorado com o estereótipo «*Transformer*» liga o transformador definido na Figura 5 (b) com o *workflow* definido nas figuras 3 e 4. A Figura 5 (a) apresenta canto superior esquerdo o mapeamento de um modelo GUI de uma visão Independente de Plataforma (PIM) para sua representação PSM, mostrado no canto inferior esquerdo. O modelo fonte é uma classe decorada com o estereótipo «*ContainerScreenArea*» e contém um atributo decorado com o estereótipo «*TextField*». O modelo alvo é uma classe que possui uma herança pela classe denominada *JPanel* (tipo de dado específico de Swing) e possui um atributo em que o tipo de dado também é específico de Swing.

A Figura 5 (b) detalha o desenvolvimento de um transformador utilizando QVT. A transformação é realizada de um modelo fonte, localizado à esquerda da Figura 5 (b), para um modelo alvo, localizado à direita da mesma figura. Da mesma maneira, o modelo alvo pode ser transformado para o modelo fonte, efetuando assim uma transformação reversa do modelo da direita para o da esquerda. Isto é realizado utilizando um relacionamento (topo e centro da Figura 5 b). Este relacionamento especifica que o objeto de nome "s" é convertido para o objeto de nome "*screenArea*" e vice-versa. O transformador da Figura 5 (b) possibilita gerar o PSM de Swing, mas não possibilita realizar uma transformação M2C. No entanto, o padrão QVT especifica que transformações M2C podem ser escritas em uma linguagem definida em QVT.

A composição de transformações possibilita desenvolver transformadores de baixo nível em níveis de dependência das características do FM, viabilizando o reuso de transformações. Isto é suficiente para permitir a manutenibilidade dos transformadores sem a necessidade de alterar o seu código fonte. No entanto, no desenvolvimento de um sistema não basta especificar uma interface gráfica de usuário. É necessário sincronizá-la com outros recursos de um projeto, como banco de dados, classes de lógica de negócio, etc. Neste caso, não é recomendado projetar os transformadores usando apenas o *workflow* de baixo nível, porque a manutenção dos transformadores seria uma tarefa complicada demais devido ao grande número de elementos necessários para configurar os transformadores. Para resolver este problema a abordagem FOMDA propõe o uso de *workflows* de alto nível.

2.4 Sincronização de Transformadores de Modelos

Workflows de alto nível são caracterizados como transformadores dirigidos por uma estrutura de transformações. Em *workflows* de alto nível o principal objetivo é sincronizar os elementos que são resultados de transformações e adicioná-los nos parâmetros de entrada dos transformadores, quando executada uma transformação "*forward*", ou nos parâmetros de saída dos transformadores quando for executada uma transformação "*reverse*". Para executar uma transformação da esquerda para a direita usando o transformador especificado na Figura 5 (b), por exemplo, é necessário adicionar o modelo da esquerda no parâmetro de entrada desse transformador. Da mesma maneira, para executar uma transformação da direita para esquerda, é necessário adicionar o modelo da direita no parâmetro de saída do mesmo transformador. Este cenário é denominado por sincronização de transformações de modelos.

O *workflow* apresentado na Figura 6 foi desenvolvido com o objetivo de sincronizar transformações e contém elementos semelhantes com os especificados no *workflow* de

baixo nível. A diferença é que o *workflow* de alto nível possui atividades de transformações ligadas com outros *workflows* de alto ou baixo nível. *Workflows* de alto nível possibilitam a composição de transformações em muitos níveis e podem ser utilizados para organizar todo um processo de transformação de modelos. O *workflow* da Figura 6 inicia por uma ação que possui o nome "*for each*". Ela é uma consulta que pesquisa no modelo de um sistema por elementos do tipo classe. Para cada elemento encontrado pela consulta (expressão *for each*), o transformador "*Gerar Layout de Cadastro*" é executado pela ação. Este transformador gera um elemento do tipo *ScreenArea* e o armazena no seu parâmetro de saída.

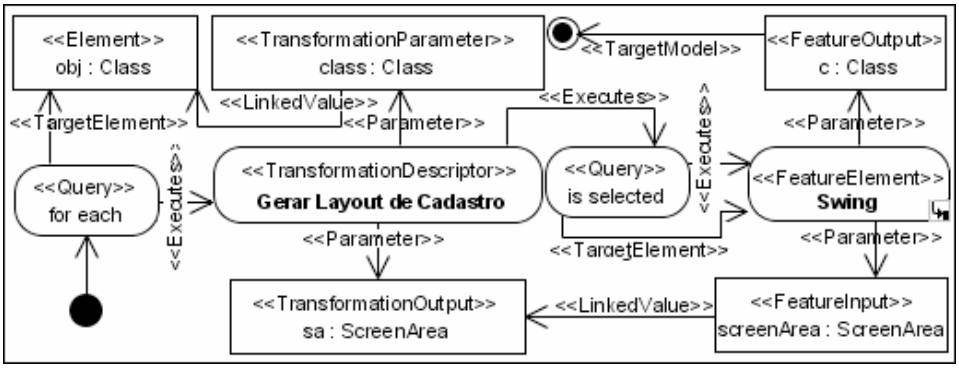

Figura 6. Sincronização de Transformações com Um Workflow de Alto Nível

Após a conclusão da transformação realizada pelo transformador "*Gerar Layout de Cadastro*" a execução de uma outra consulta, que possui a expressão "*is selected*", é chamada por este transformador. Esta consulta testa se a características "*Swing*" está selecionada no FM. Caso esteja selecionada, a atividade de transformação "*Swing*" é executada. Esta atividade é vinculada à característica "*Swing*" do FM por causa do uso do estereótipo «*FeatureElement*». Ela possui um estado de sub atividade que liga a atividade "*Swing*" com o *workflow* de baixo nível especificado nas figuras 3 e 4. Quando atividade de transformação "*Swing*" é executada, o valor contido no parâmetro de entrada "*screenArea*" é repassado para o parâmetro seguinte ao estado inicial do *workflow* que está ligado à atividade "*Swing*" pelo estado de sub atividade. Portanto, o parâmetro de nome "*screenArea*" mostrado na Figura 3 recebe esse valor.

Além disso, o *workflow* de alto nível possibilita ao projetista definir regras para execução de transformadores de modelos. Estas regras podem, por exemplo, restringir o acesso a determinados transformadores por tipos de usuários. Este *workflow* possibilita ao projetista destinar tarefas para determinados tipos de usuário como projetistas de interface gráfica de usuário, modelador de mapeamento objeto-relacional, etc. As regras para execução de transformadores de modelos não são detalhadas neste documento.

2.5 A Ferramenta WorkCASE Toolkit

A ferramenta WorkCASE Toolkit[1] foi desenvolvida para suportar a abordagem FOMDA. Usando o toolkit o projetista de transformações pode especificar FMs e selecionar as características do modelo, marcando a plataforma alvo que deve ser utilizada para desenvolver um sistema. Transformadores TMD podem ser acrescentados dinamicamente na ferramenta. A ferramenta também permite especificar a composição de transformadores e os *workflows*.

[1] www.workcase.com.br

O modelo de um sistema especificado por um projetista de uma aplicação é um documento no formato XMI [18]. Este documento contém meta-dados UML representados no formato XML. Esta ferramenta utiliza um Parser de XMI para converter os meta-dados, que estão no formato XML, para elementos que são instâncias do MOF. Isto permite que os transformadores manipulem elementos do MOF e não o XMI, o que possibilita o reuso de transformadores em caso de troca de versões do XMI no documento do modelo.

Além de suportar a abordagem FOMDA, a WorkCASE Toolkit possui mecanismos para estender suas funcionalidades. Ela é uma ferramenta preparada para o desenvolvimento de linguagens de modelagem específicas (DSMLs). A UML provê mecanismos de extensão que podem ser usados para "converter" elementos especificados em uma DSML em elementos UML, bem como a conversão reversa. A ferramenta WorkCASE Toolkit possibilita que um especialista em MDA adicione interceptadores de *tags* e estereótipos para converter elementos UML que estão marcados com restrições de um perfil em sua representação natural da DSML.

Este toolkit também disponibiliza um *framework* para desenvolvimento de transformadores TMD. A linguagem utilizada para codificar os transformadores TMD é uma implementação comercial na linguagem de programação *java* do meta-modelo FOMDA de transformações, chamada de API WCT. Esta API permite que sejam desenvolvidos transformadores com foco para a composição de transformações descritas pela abordagem FOMDA.

3. Estudo de Caso

Esta seção apresenta um estudo de caso realizado na empresa TKnowledge[1], que exemplifica a geração de Modelos Específicos de Plataformas (PSMs) para três plataformas distintas. O problema consiste na transformação de um modelo especificado em alto nível para modelos específicos de Swing, AWT e JSTL. O modelo em consideração especifica requisitos para interfaces gráficas de usuário (GUI) utilizando o *GUI Profile* [15] e também de uma classe que contém propriedades que estão mapeadas para campos desta GUI. Este modelo pode ser visto na Figura 7. No canto esquerdo dessa figura é mostrada a classe *UserImpl* que define propriedades de um usuário e o modelo *UserImplCadastreForm* que possui um campo de texto denominado *usernameTextField* o qual a propriedade *username* da classe *UserImpl* está mapeada.

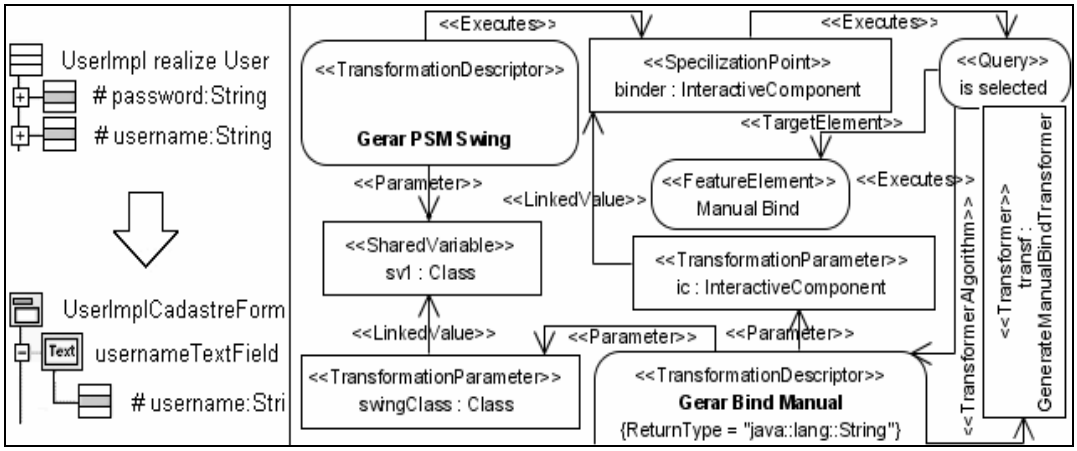

Figura 7. Modelo de Interface Gráfica de Usuário e Workflow de Baixo Nível Incrementado para Aplicar Bind Manual

O objetivo do estudo é transformar o elemento *UserImplCadastreForm* em três classes distintas (PSMs), uma dependente de *Swing*, outra de *AWT* e outra de *JSTL*. Nos exemplos utilizados no estudo de caso foram somente realizadas transformações "*forward*" para os PSMs de Swing e AWT. Além disso, para cada PSM a geração de dois tipos de *bind* é realizada: um utilizando o componente *Commons Binder* e outro *Manual Bind*. A Figura 8 (a) apresenta o mapeamento de um código dependente da característica *Swing* para o *bind* da propriedade *username* utilizando a característica *Manual Bind*. A Figura 8 (b) apresenta este mapeamento utilizando a característica *Commons Binder*. Analisando os aspectos de variabilidade entre os códigos definidos na Figura 8, se percebe que os dois mapeamentos são muito diferentes, coincidindo na necessidade pelo atributo *user* que está sublinhado na figura.

a) Manual Bind

```
protected JTextField
  getUsernameTextField(){
  ... return usernameTextField;
}
UserImpl user = new UserImpl();
public void bind(){
  this.getUsernameTextField().
     setText(this.user.getUsername());
}
public void reverseBind(){
  this.user.setUsername(this.getUser
     nameTextField().getText());
}
```

b) Commons Binder

```
Binder binder = new Binder();
UserImpl user = new UserImpl();
protected JTextField getUsername
  TextField(){
  if (usernameTextField == null) {
      usernameTextField = new
         JTextField();
      this.binder.addBindProperty
         (this.user,this.username
         TextField,"username");
  }
  return usernameTextField;
}
```

Figura 8. Variabilidade em Código Fonte por Características Alvo

Um transformador TMD foi desenvolvido para gerar cada um dos mapeamentos definidos na Figura 8. Um transformador TMD na WorkCASE Toolkit é uma classe escrita com a linguagem de programação *java* que estende a classe *AbstractTransformer*. Algoritmos para especificar transformações "*forward*" devem ser escritas na operação *doTransformation* e algoritmos para transformações "*reverse*" devem ser escritas na operação *doReverseTransformation*. O *workflow* mostrado na Figura 7 especifica o projeto de transformadores utilizados para aplicar as transformações requeridas. Ao transformador "*Gerar PSM Swing*" foi adicionado um ponto de especialização de nome "*binder*". O transformador "*Gerar Bind Manual*" foi ligado neste ponto com o uso de ações decoradas com o estereótipo «*Query*». O objeto decorado com o estereótipo «*Transformer*» liga um transformador TMD com a atividade de transformação "*Gerar Bind Manual*".

Um trecho do código fonte do transformador "*Gerar PSM Swing*" é mostrado na Figura 9. A operação *getBodyCode* (linhas 8-12) é usada para retornar a representação em código do corpo da operação *getUsernameTextField*, como exemplificado na Figura 8 (b). Se o elemento "*sa*" que é recebido como parâmetro pela operação *getBodyCode* é instância de *InteractiveComponent*, a operação *getBindCode* é chamada (linhas 1-7). Nesta operação o ponto de especialização de nome "*binder*" é recuperado do transformador "*Gerar PSM Swing*" (linha 2). O objeto "*ic*" é adicionado neste ponto de especialização (linha 4). Este ponto é executado na linha 5 do algoritmo do transformador. O resultado da execução do ponto de especialização é uma string que é retornada pela operação *getBindCode*. Esta string é concatenada com a string que é inserida no corpo da operação *getUsernameTextField* (linha 12).

A Figura 9 exemplificou parte da transformação para gerar um PSM de Swing. Neste PSM deve estar representado ou o mapeamento da Figura 8 (a) ou (b). O transformador especificado na Figura 10 gera o código da Figura 8 (a) e o transformador especificado na Figura 11 gera o código da Figura 8 (b). O que determina o código que estará presente na classe Swing é a execução do ponto de especialização de nome "*binder*". A execução do ponto de especialização executa também uma consulta que verifica qual das características "*Manual Bind*" ou "*Commons Binder*" está selecionada no FM e, de acordo com a seleção, a consulta chama a execução de um dos transformadores para *bind*.

```
1. private String getBindCode(InteractiveComponent ic){
2.    SpecializationPoint sp = getSpecializationPoint("binder");
3.    try {
4.      sp.setElement(ic);
5.      return (String) sp.execute();
6.    } catch (Exception e) {}
7.    return null;}
8. private String getBodyCode(ScreenArea sa){
9.    String str="";
10.   str+="this."+sa.getInstanceName()+" = new "+
                        getScreenAreaType(sa).getName()+"();";
11.   if (sa instanceof InteractiveComponent){
12.     str+=getBindCode((InteractiveComponent) sa);
```

Figura 9. Chamada de Execução de um Ponto de Especialização em um Transformador

O mapeamento da Figura 8 (a) especifica que é necessário pouco código para definir um componente gráfico. Este componente é definido no corpo da primeira operação denominada *getUsernameTextField*. Já nas operações *bind* e *reverseBind*, que especificam o *bind* para o componente definido na primeira operação, mais código é necessário. Então, o transformador da Figura 10 retorna uma string vazia, mas define as operações necessárias (linhas 7 e 8) e adiciona nestas operações o código requerido pelo mapeamento de *Manual Bind* (linhas 9 e 10). O Mapeamento da Figura 8 b determina a criação do atributo de nome "*binder*" e a especificação de código no corpo da operação *getUsernameTextField*. O atributo de nome "*binder*" é adicionado na classe Swing pelo transformador da Figura 11 (linha 7). A especificação do código respectivo ao *bind* no corpo do componente Swing é criada na linha 8. O retorno deste transformador é uma string contendo o código requerido pelo mapeamento da Figura 8 (b).

```
1. public Object doTransformation() throws Exception{
2.    InteractiveComponent ic = (InteractiveComponent)
   getParameter("interactiveComponent");
3.    Class swingClass =  (Class) getParameter("swingClass");
4.    Property p =(Property) ic.getMappedProperty();
5.    Class mappedClass = (Class) ic.getMappedPropertyClass();
6.    swingClass.addElement(getPAttribute(mappedClass));
7.    Operation bind = (Operation) getBindOp(swingClass);
8.    Operation rbind = (Operation) getRBindOp(swingClass);
9.  bind.setBody(bind.getBody()+'\n'+"this.get"+firstUpperCase(ic.getInstanceNa
   me())+"().setText(this."+firstLowerCase(mappedClass.getName())+".get"+first
   UpperCase(p.getName())+"());");
10. rbind.setBody(bind.getBody()+'\n'+"this."+firstLowerCase(mappedClass.getNam
   e())+".set"+firstUpperCase(p.getName())+"(this.get"+firstUpperCase(ic.getIn
   stanceName())+"().getText());");
11. return "";}
```

Figura 10. Transformador para Manual Bind

```
1. public Object doTransformation() throws Exception{
2.   InteractiveComponent ic = (InteractiveComponent)
   getParameter("interactiveComponent");
3.   Class swingClass = (Class) getParameter("swingClass");
4.   Property p =(Property) ic.getMappedProperty();
5.   Class mappedClass = (Class) ic.getMappedPropertyClass();
6.   swingClass.addElement(getPAttribute(mappedClass));
7.   swingClass.addElement(getBindAttribute(mappedClass));
8.   String str=""+"this.binder.addBindProperty(this."+firstLowerCase
   (mappedClass.getName())+ +,this."+ic.getInstanceName()+",\""+
   p.getName()+"\");";
9.   return str;}
10.public Attribute getPAttribute(Class pClass){
11.  Attribute at = new Attribute(pClass);
12.  at.setName(firstLowerCase(pClass.getName()));
13.  at.setType(pClass);
14.  at.setBody("new "+pClass.getName()+"();");
15.  return at;}
```

Figura 11. Transformador para Commons Binder

3.1 Troca de Plataformas Alvo

A troca da plataforma alvo Swing para Awt requer o desenvolvimento de um novo transformador para gerar código dependente de *Awt*. No entanto, o mesmo transformador para Swing pode ser adaptado para gerar código para Awt ou para qualquer API gráfica *java*. Basta que, para isso, os tipos de dados dos componentes sejam também gerados por transformadores combinados. A linha 10 (Figura 9) do algoritmo do transformador "*Gerar Swing PSM*" contém uma chamada para a operação *getScreenAreaType* mostrada no topo da Figura 12. Esta operação retorna um objeto do tipo *Type* que é uma abstração de um tipo de dado definido na UML. Esta operação chama a execução do ponto de especialização de nome "*saType*" especificado na Figura 12 (linhas 2 à 5). O transformador "*Gerar Swing PSM*" foi denominado "*Gerar Java GUI PSM*" porque pode gerar interfaces gráficas *java*, bastando que transformadores sejam adicionados no ponto de especialização de nome "*saType*".

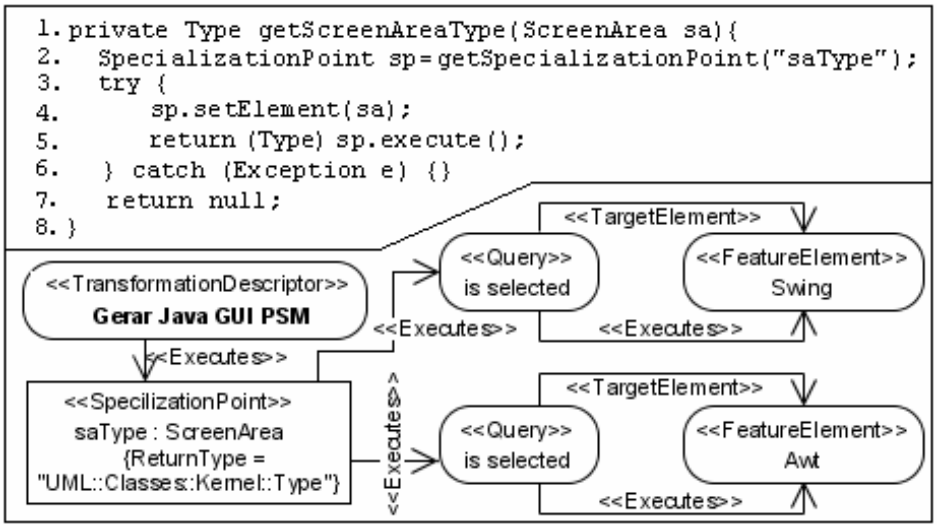

Figura 12. Workflow e Ponto de Especialização que Permitem a Execução de Um Transformador de Interfaces Gráficas de Usuário para Modelos Alvo Específicos de Java

Na troca da plataforma Swing para JSTL não foi possível reutilizar o transformador "*Gerar Java GUI PSM*". O motivo é que o código para usar JSTL mistura *tags* HTML, JSP e JSTL, que em nada tem a ver com o código Swing, AWT ou outra API gráfica *java*. A impossibilidade de reuso neste caso é natural devido às diferenças entre os PSMs. Para transformar um modelo fonte para as características JSTL, JSP e Jasper foi desenvolvido um transformador denominado "*Gerar GUI WEB PSM*". O resultado obtido foi o mesmo que para Swing e AWT, ou seja, foi possível reutilizar os transformadores relacionados com JSP, JSTL, internacionalização, *bind* e validação para interfaces gráficas WEB.

3.2 Avaliação de Manutenibilidade e Reuso de Transformadores

O estudo de caso possibilitou avaliar a manutenibilidade e o reuso dos transformadores definidos pela abordagem FOMDA. Foi possível identificar que transformadores são reutilizáveis em certos níveis de independência de plataformas. O nível de independência é relacionado com a linguagem que o PSM é mapeado. Com o estudo de caso foi possível identificar que para as linguagens Java e HTML não é viável desenvolver um único transformador, como para o caso de Swing e Awt ou de JSTL e JSP. Talvez um transformador genérico para interfaces gráficas nas linguagens de programação *java* e *c++* seja viável, tendo assim que definir pontos de especialização para herança e algumas palavras chaves que diferem entre as linguagens. Ainda assim, o custo/benefício não é um fator encorajador.

4. Trabalhos Relacionados

Foram identificadas algumas abordagens com foco em reuso de transformações. Na documentação da ferramenta *pure::variants* [21] foi possível identificar que a composição de transformações apenas é feita no código dos transformadores. A abordagem de Almeida et al. [14] é interessante para transformações relacionadas ao aspecto de composição, no entanto não foi possível testá-la em uma ferramenta para MDA. Wagelaar et. al. especificaram a execução de transformações para linguagens de modelagem de domínio específico (DSMLs) [7]. Isto não é possível de ser feito usando QVT [19]. A abordagem FOMDA permite a execução de transformadores de elementos de DSMLs. Balogh et. al. demonstra como efetuar composição de transformações baseadas em grafos [1] com o sistema VIATRA2. A mesma composição é possível de ser feita em QVT. A abordagem FOMDA não especifica transformações baseadas em grafo, mas permite utilizar abordagens deste tipo para efetuar transformações.

Outras abordagens para MDA podem ser adaptadas como transformadores na abordagem FOMDA, bem como é possível para QVT. FOMDA e QVT podem ser utilizadas em conjunto para combinar transformações e o uso de uma abordagem não anula o uso da outra. Usando a abordagem FOMDA é possível visualizar o fluxo de execução das transformações, o que não é possível usando QVT. Além disso, a abordagem FOMDA permite a execução de transformações com base no FM (que não é suportado em QVT). Esta abordagem traz contribuições importantes para transformação de modelos, o que sugere a sua inclusão no padrão QVT.

5. Contribuições, Restrições e Trabalhos Futuros

Este trabalho apresentou uma solução para o reuso e a manutenção de transformadores em decorrência de troca de plataformas alvo utilizadas para desenvolver sistemas. O reuso ocorre em todos os aspectos identificados por Olsen et. al em [17]. A manutenção

dos transformadores é realizada de forma dinâmica, combinando transformadores de modelos em quatro etapas definidas na abordagem FOMDA. Esta solução foi implantada no desenvolvimento de transformadores em um ambiente corporativo e permite à corporação que os transformadores sejam adaptados para as inevitáveis mudanças nas plataformas utilizadas por ela.

Foi identificado no estudo de caso que para plataformas que são muito diferentes, a transformação de um PIM para os PSMs pode ocorrer sem reuso algum dos transformadores. Isto é natural, para os casos em que os PSMs são muito diferentes. Ainda assim, a impossibilidade de reuso é um fator que dificulta o desenvolvimento e a manutenção de transformadores de modelos. Para esta situação a abordagem de Willink é bastante interessante e define uma linguagem gráfica de transformação de modelos chamada de UMLX [16], muito semelhante à especificação para mapeamentos definida no padrão QVT. Um estudo precisa ser realizado para avaliar se é possível efetuar transformações como as exemplificadas no estudo de caso usando a UMLX, para, por fim, avaliar os aspectos de reuso de transformações dessa abordagem.

A ferramenta WorkCASE Toolkit foi utilizada como ferramenta MDA para executar as transformações descritas neste documento. No entanto, não foi possível executar a transformação especificada com QVT porque para executá-la é necessário o desenvolvimento de um interpretador para QVT. Pretende-se desenvolver este interpretador nesta ferramenta.

Por fim, as ações definidas nos *workflows* da abordagem FOMDA devem ser modificadas para elementos definidos pela MDA. Estes elementos estão disponíveis em um padrão usado para especificar ações em modelos de sistemas, denominado *Action Semantics*, que pode ser utilizado na abordagem FOMDA para especificar ações.

Referências

[1] A. Balogh, D. Varro. Pattern composition in graph transformation rules. First European Workshop on Composition of Model Transformations. 2006, pp. 31-37.

[2] B. Vanhooff, D. Ayed, and Y. Berbers. A Framework for Transformation Chain Development Processes. First European Workshop on Composition of Model Transformations. 2006, pp. 3-8.

[3] B. Selic. Model-Driven Development: Its Essence and Opportunities. In Proc. of Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing. Gyeongju, Korea. 2006. pp. 313-319.

[4] 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.

[5] C. Kang et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-21). Software Engineering Institute, Carnegie Mellon University. Pittsburg, PA. November, 1993.

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

[7] D. Wagelaar. Blackbox Composition of Model Transformations using Domain-Specific Modelling Languages. First European Workshop on Composition of Model Transformations. 2006, pp. 15-19.

[8] E. Willink. UMLX: A graphical transformation language for MDA. In proceedings of Model-Driven Architecture: Foundations and Applications 2003. pp 13-24.

[9] F. Basso. Features-Oriented Model-Driven-Architecture: Uma Abordagem para MDD. Master Thesis. 29 March 2006. Pontifícia Universidade Católica do Rio Grande do Sul. Department of Computer Science. 2006.

[10] F. Basso, T. Oliveira, L. Becker. Using the FOMDA Approach to Support Object-Oriented Real-Time Systems Development; In Proc. of Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing. Gyeongju, Korea. 2006. pp. 374-381.

[11] G. Baixauli, M. Laguna. MDA e Ingeniería de Requisitos para Líneas de Producto. II Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones (DSDM 05). 2005. pp 11-20.

[12] G. Boas. From the Workfloor: Developing Workflow for the Generative Model Transformer. OOPSLA 2005.

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

[14] 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.

[15] K. Blankenhorn. A UML Profile for GUI Layout. Master Thesis. 23 May 2004. Universisty of Applied Sciences Furtwangen. Department of Digital Media. 2004.

[16] 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.

[17] G. Olsen, J. Aagedal, J. Oldevik. Aspects of Reusable Model Transformations. . First European Workshop on Composition of Model Transformations. 2006, pp. 21-26.

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

[19] OMG, MOF 2.0 Query View and Transformation specifications version 2.0, January 2007, OMG document . Available at <http://www.omg.org/cgi-bin/doc?ptc/2005-11-01>

[20] T. Oliveira et al. Enabling Model Driven Product Line Architectures. In: Second European Workshop on Model Driven Architecture (ECMDA-FA); Canterbury, England, 2004.

[21] Technical White Paper. Variant Management With Pure Variants. April 2007. Available < http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf>

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