UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

CENTRO DE CIÊNCIAS EXATAS E DA TERRA

PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

EDMILSON BARBALHO CAMPOS NETO

UM MÉTODO PARA DESENVOLVIMENTO DE ABORDAGENS GENERATIVAS COM COMPOSIÇÃO DE LINGUAGENS ESPECÍFICAS DE DOMÍNIO

NATAL/RN

2013

EDMILSON BARBALHO CAMPOS NETO

UM MÉTODO PARA DESENVOLVIMENTO DE ABORDAGENS GENERATIVAS COM COMPOSIÇÃO DE LINGUAGENS ESPECÍFICAS DE DOMÍNIO

Dissertação de mestrado submetida à Coordenação do Programa de Pós-Graduação em Sistemas e Computação, do Centro de Ciências Exatas e da Terra, da Universidade Federal do Rio Grande do Norte, como parte dos requisitos para obtenção de título de Mestre em Sistemas e Computação.

Orientador: Uirá Kulesza, Prof. Dr.

NATAL/RN

2013

UFRN / Biblioteca Central Zila Mamede
Catalogação da Publicação na Fonte

Campos Neto, Edmilson Barbalho.
Um método para desenvolvimento de abordagens generativas com composição de linguagens específicas de domínio. / Edmilson Barbalho Campos Neto. – Natal, RN, 2013.
111 f.: il.

Orientador: Prof. Dr. Uirá Kulesza.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Programa de Pós-Graduação em Sistemas e Computação.

1. Engenharia de software automatizada – Dissertação. 2. Desenvolvimento generativo - Dissertação. 3. Linguagens específicas de domínio - Composição - Dissertação. 4. Engenharia de linha de produto de software – Dissertação. I. Kulesza, Uirá. II. Universidade Federal do Rio Grande do Norte. III. Título.

RN/UF/BCZM CDU
004.41

EDMILSON BARBALHO CAMPOS NETO

Um Método para Desenvolvimento de Abordagens Generativas com Composição de Linguagens Específicas de Domínio

Esta Dissertação foi julgada adequada para a obtenção do título de mestre em Sistemas e Computação e aprovado em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

Prof. Dr. Uirá Kulesza – UFRN
Orientador

Prof. Dr. Martin Alejandro Musicante – UFRN
Coordenador do Programa

Banca Examinadora

Prof. Dr. Uirá Kulesza – UFRN
Presidente

Prof. Dr. Gibeon Soares de Aquino Junior – UFRN

Prof. Dr. Franklin de Souza Ramalho – UFCG

Agosto, 2013

A todos que depositaram confiança e estiveram na
torcida pelo sucesso deste trabalho.

AGRADECIMENTO

Em agradecimento, deixo algumas palavras para que possam atingir a essência daqueles que contribuíram para a conclusão deste trabalho.

Meus sinceros agradecimentos à Uirá Kulesza, meu orientador neste projeto, por suas incessantes mobilizações pela realização deste trabalho e pelos créditos concedidos a esta tencionada produção.

Aos companheiros de pesquisa, Marília, Fellipe e Daniel, que me acolheram no grupo em meu ingresso e que durante esta caminhada foram parceiros, dividindo comigo muitas horas de trabalho.

Aos colegas citados em texto, que de forma direta ou indireta somaram esforços com o meu trabalho, tenha sido nas escritas partilhadas de artigos, nas implementações em parcerias ou até mesmo nos *feedbacks* nas apresentações.

Em especial, agradeço a Deus, seus anjos e a minha família e aos amigos apoiadores nesta caminhada.

E por fim, agradeço também aos não citados, mas que mesmo influíram neste trajeto, contribuindo com o seguimento e finalização deste trabalho.

Com todos vocês, partilho da felicidade desta conquista!

Uma existência é um ato.

[...]

Um triunfo — uma aquisição.

(André Luiz – psicografia de Chico Xavier, em Nosso Lar)

RESUMO

A utilização de linguagens específicas de domínios para o desenvolvimento de sistemas de software tem se tornado cada vez mais comum. Elas propiciam um aumento da expressividade do domínio, elevando o seu nível de abstração através de facilidades para geração de modelos ou códigos de baixo-nível, que aumentam assim a produtividade do desenvolvimento de sistemas. Como consequência, métodos para o desenvolvimento de linhas de produtos de software e famílias de sistemas também têm proposto a utilização de linguagens específicas de domínio (*domain-specific languages* – DSLs). Estudos recentes têm investigado os limites de expressividade do modelo de *features*, e propondo o uso de DSLs em sua substituição ou complemento. Contudo, em projetos complexos, uma única DSL muitas vezes é insuficiente para representar as diferentes visões e perspectivas do desenvolvimento, sendo necessário trabalhar com múltiplas DSLs. Com isso surgem novos desafios, tais como a gerência de consistência entre as DSLs, e a necessidade de métodos e ferramentas que ofereçam suporte ao desenvolvimento com múltiplas DSLs. Ao longo dos últimos anos, diversas abordagens têm sido propostas para o desenvolvimento de abordagens generativas, entretanto, nenhuma delas considera questões relacionadas à composição de DSLs. Assim, visando abordar tal problemática, os principais objetivos desta dissertação são: (i) investigar a adoção do uso integrado de modelos de *features* e DSLs tanto na engenharia de domínio quanto de aplicação de desenvolvimento de abordagens generativas; (ii) propor um método para o desenvolvimento de abordagens generativas com composição de DSLs; e (iii) investigar e avaliar o uso de tecnologias atuais de engenharia dirigida por modelos na implementação de estratégias de integração entre modelos de *features* e composição de DSLs.

Palavras-chave: engenharia de software automatizada; desenvolvimento generativo; composição de linguagens específicas de domínio; engenharia de linha de produto de software.

ABSTRACT

The software systems development with domain-specific languages has become increasingly common. Domain-specific languages (DSLs) provide increased of the domain expressiveness, raising the abstraction level by facilitating the generation of models or low-level source code, thus increasing the productivity of systems development. Consequently, methods for the development of software product lines and software system families have also proposed the adoption of domain-specific languages. Recent studies have investigated the limitations of feature model expressiveness and proposing the use of DSLs as a complement or substitute for feature model. However, in complex projects, a single DSL is often insufficient to represent the different views and perspectives of development, being necessary to work with multiple DSLs. In order to address new challenges in this context, such as the management of consistency between DSLs, and the need to methods and tools that support the development with multiple DSLs, over the past years, several approaches have been proposed for the development of generative approaches. However, none of them considers matters relating to the composition of DSLs. Thus, with the aim to address this problem, the main objectives of this dissertation are: (i) to investigate the adoption of the integrated use of feature models and DSLs during the domain and application engineering of the development of generative approaches; (ii) to propose a method for the development of generative approaches with composition DSLs; and (iii) to investigate and evaluate the usage of modern technology based on models driven engineering to implement strategies of integration between feature models and composition of DSLs.

Keywords: automated software engineering; generative development; composition of domain-specific languages; software product line engineering.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE LISTAGENS

LISTA DE ABREVIATURAS E SIGLAS

BNF	*Backus-Naur Form*
CRD	*Completely Randomized Design*
DSL	*Domain-Specific Language*
DSSA	*Domain-Specific Software Architecture*
EA	Engenharia de Aplicação
ED	Engenharia de Domínio
EMF	*Eclipse Modeling Framework*
ESE	Engenharia de Software Experimental
FM	*Feature Model*
FODA	*Feature Oriented Domain Analysis*
GPL	*General Purpose Language*
JPDL	jBPM *Process Definition Language*
IDE	*Integrated Development Environment*
LS	*Latin Square*
LPS	Linha de Produto de Software
MAPLE	*Model-driven Aspect-oriented Product Line Engineering*
MDD	*Model-Driven Development*
MVC	*Model-View-Controller*
M2M	*Model-To-Model*
M2T	*Model-To-Text*
OFBIz	*Open for Business*
PLE	*Product Line Engineering*
QVTc	*Core QVT*
QVTd	*Declarative QVT*

QVTo	*Operational QVT*
RCBD	*Randomized Complete Block Design*
SQL	*Structured Query Language*
UML	*Unified Modeling Language*
VML	*Variability Modeling Language*
XMI	*XML Metadata Interchange*
XML	*eXtensible Markup Language*

SUMÁRIO

1. INTRODUÇÃO

O desenvolvimento de sistemas de software utilizando linguagens específicas de domínio (*Domain-Specific Languages* – DSLs) tem aumentado nos últimos anos. Elas ampliam a expressividade do domínio ao focar em soluções específicas para o mesmo, objetivando a resolução de problemas próprio do cenário (FOWLER e PARSONS, 2013), diferentemente das linguagens de propósito geral (*General Purpose Languages* – GPLs), tais como Java e C++, que provêm soluções mais amplas, genéricas, nem sempre tão otimizadas para o desenvolvimento dentro de um domínio específico. A adoção de DSLs aumenta o nível de abstração e traz facilidades para geração de modelos ou código de níveis inferiores, trazendo assim o potencial de aumentar a produtividade do desenvolvimento de sistemas em diferentes domínios (LOCHMANN e BRÄUER, 2007). Além disso, sabe-se que sucesso em um projeto de desenvolvimento requer efetiva colaboração dos seus *stakeholders*, além de flexível e coerente conceituação do domínio do problema (HESSELLUND, CZARNECKI e WASOWSKI, 2007). No contexto de projetos mais complexos, entretanto, uma única DSL pode ser insuficiente para tratar diferentes visões e perspectivas de modelagem de um domínio de software, demandando assim a utilização de múltiplas DSLs durante o seu desenvolvimento. A principal consequência é o aumento do risco de perda de consistência entre elementos dos modelos (MENS, STRAETEN e D'HONDT, 2006), que implica na necessidade de uma investigação mais apurada desses aspectos, afim que se possam encontrar soluções mais eficazes. Manter a consistência entre vários modelos é um dos grandes desafios da composição de DSLs, que requer a adoção de métodos e ferramentas de suporte ao desenvolvimento.

Ao longo dos últimos anos, métodos para o desenvolvimento de abordagens generativas têm sido propostos os quais incorporam a utilização de DSLs (CZARNECKI e EISENECKER, 2000; GREENFIELD, SHORT, *et al.*, 2004). Estes métodos buscam o desenvolvimento de linhas de produto de software (LPS) ou famílias de sistemas pertencentes a um mesmo domínio (CLEMENTS e NORTHROP, 2011; WEISS e LAI, 1999), promovendo a reutilização em larga escala. A ideia de reutilização está associada não apenas ao reuso no nível de código-fonte, mas também e, principalmente, ao reuso de uma arquitetura central

formada por um núcleo de artefatos, que atendem aplicações pertencentes a um mesmo domínio ou segmento de mercado. De acordo com (POHL, BÖCKLE e LINDEN , 2005), o desenvolvimento de uma LPS pode ser definida basicamente em duas fases: (i) Engenharia de Domínio (ED) – que envolve a delimitação do escopo da LPS e desenvolvimento dos artefatos reusáveis; e (ii) Engenharia de Aplicação (EA) – responsável pela geração dos produtos da LPS, através do reuso dos artefatos gerados durante a ED. Assim, em um processo típico de desenvolvimento generativo de LPS (CZARNECKI e EISENECKER, 2000), após a especificação e implementação da arquitetura de LPS na ED, ocorre a etapa de EA. Durante esse estágio, os artefatos de implementação produzidos anteriormente são compostos e integrados para gerar uma instância (produto) da LPS. Esse processo é também conhecido como derivação de produto (DEELSTRA, SINNEMA e BOSCH, 2005).

Uma LPS pode ser definida como uma família de sistemas específicos de um segmento do mercado, que tipicamente são especificadas, modeladas e implementadas conforme suas características (*features*) comuns e variáveis. Uma *feature* (CZARNECKI e HELSEN, 2006) pode ser uma propriedade ou uma funcionalidade do sistema que seja relevante para algum *stakeholder* e usada para discriminar características comuns ou variáveis entre os sistemas em uma LPS. Tradicionalmente, modelos de *features* são utilizados para representar os elementos de alto nível de uma LPS em uma abordagem generativa. Eles formam uma representação compacta das *features* dos possíveis produtos de uma LPS, assim como suas restrições e são usados para especificar produtos a serem derivados. Trabalhos recentes da comunidade, tal como (VOELTER e VISSER, 2011), têm investigado os limites de expressividades dos modelos de *features* e propõe o uso de DSLs de forma complementar no desenvolvimento de abordagens generativas. Contudo, nenhum desses trabalhos propõe uma abordagem para desenvolver e compor DSLs no contexto de abordagens generativas.

1.1. Problema Abordado e Limitações de Trabalhos Relacionados

Embora algumas abordagens tenham sido propostas para o desenvolvimento de abordagens generativas (GREENFIELD, SHORT, *et al.*, 2004; CZARNECKI e EISENECKER, 2000; GAMMA, HELM, *et al.*, 1994; WEISS e LAI, 1999), nenhuma

delas propõem ou lidam explicitamente com a integração e composição de múltiplas DSLs. Por outro lado, trabalhos de pesquisa recentes têm abordado a questão da consistência entre modelos como objeto de estudo (MENS, STRAETEN e D'HONDT, 2006; BÉZIVIN e JOUAULT, 2005), porém poucos são relacionados à composição de linguagens específicas de domínio. Hessellund *et al*. (HESSELLUND, 2009; HESSELLUND e LOCHMANN, 2009) propõem um método sistemático para desenvolvimento envolvendo a composição de múltiplas DSLs, embora de forma não integrada a um método de desenvolvimento de abordagens generativas.

Desse modo, este trabalho investiga o problema de desenvolvimento de abordagens generativas envolvendo a integração do modelo de *features* com múltiplas linguagens específicas de domínio, que carece de estudos que apresentem metodologias e ou técnicas para condução de um desenvolvimento com composição de múltiplas DSLs em tais abordagens.

1.2. Objetivos do Estudo e Questões de Pesquisa

Os principais objetivos desta dissertação são: (i) investigar a adoção do uso integrado de modelos de *features* e linguagens específicas de domínio tanto na engenharia de domínio quanto de aplicação de desenvolvimento de abordagens generativas; (ii) propor um método para o desenvolvimento de abordagens generativas com composição de DSLs; e (iii) investigar e avaliar o uso de tecnologias atuais de engenharia dirigida por modelos na implementação de estratégias de integração entre modelos de *features* e composição de linguagens específicas de domínio.

O método proposto nesta dissertação para desenvolvimento de abordagens generativas resulta da investigação de outras abordagens existentes. Ele é implementado e avaliado através do uso de tecnologias de engenharia dirigida por modelos. Tal escolha foi realizada devido à grande atenção que os trabalhos de pesquisa e da indústria têm dado recentemente para tais tecnologias, incluindo ferramentas para a engenharia de linhas de produto de software.

Assim, as questões de pesquisa que guiaram o desenvolvimento deste trabalho, de forma a buscar atingir tais objetivos, foram:

QP1. De que forma modelos de *features* e linguagens específicas de domínio podem ser usados de forma integrada durante o desenvolvimento de abordagens generativas?

QP1.1. Qual o papel e utilidade do modelo de *features* em uma abordagem generativa que envolva a composição de múltiplas linguagens específicas de domínio durante a engenharia de domínio?

QP1.2. Que estratégias de derivação podem ser utilizadas para abordagens generativas que envolvam a integração do modelo de *features* e múltiplas linguagens específicas de domínio? Quais as vantagens e desvantagens de tais estratégias?

QP2. Como implementar abordagens generativas que envolvam a composição de múltiplas linguagens específicas de domínio usando tecnologias atuais de engenharia dirigida por modelos?

QP2.1. Como a composição de linguagens específicas de domínio pode ser especificada e implementada durante a engenharia de domínio?

QP2.2. Como estratégias de derivação de produtos/sistemas de abordagens generativas que envolvam a composição de linguagens específicas de domínio podem ser implementadas na engenharia de aplicação?

1.3. Trabalho Proposto e Metodologia Adotada

Visando responder as questões de pesquisa desta dissertação, este trabalho propõe um método para o desenvolvimento de abordagens generativas que envolva a integração de modelos de *features* e linguagens específicas de domínio na Engenharia de Domínio e Aplicação. O método se baseia em outras abordagens existentes na literatura, que tratam ao menos parcialmente os problemas investigados. Além disso, o método apresenta diferentes alternativas de integração dos modelos de *features* com linguagens específicas de domínio, apresentando estratégias alternativas de derivação de produtos em uma LPS ou família de sistemas, usando composição de DSLs.

Para responder as questões de pesquisa apresentadas anteriormente e atingir os objetivos desta dissertação, inicialmente definimos um método para desenvolvimento de abordagens generativas com composição de DSLs e então

realizamos um estudo exploratório para avaliação do método proposto. O estudo investigou a aplicação do método em um domínio real e resultou no projeto e implementação de uma abordagem generativa para modelagem de experimentos controlados e geração de *workflows* para seus participantes, através da adoção de múltiplas DSLs. Além disso, foram derivados produtos da LPS utilizando diferentes estratégias propostas pelo método.

1.4. Contexto

Este trabalho está inserido em um contexto mais amplo de estudo que envolve a participação de outros pesquisadores do nosso grupo de pesquisa[1] e resulta em uma abordagem dirigida por modelos para definição e execução de experimentos controlados em engenharia de software. A abordagem, desenvolvida por (FREIRE, ACCIOLY, *et al.*, 2013), propõe o uso de técnicas de engenharia dirigida por modelo e linguagens específicas de domínio para gerar ambientes automatizados que facilitem o planejamento e a execução de experimentos controlados em engenharia de software.

Esta dissertação explora o desenvolvimento da abordagem citada dentro do contexto de um estudo exploratório que buscou avaliar o método proposto. DSLs foram criadas utilizando tecnologias dirigidas a modelos e compostas para definir e modelar processos de experimentação com suas respectivas atividades, papéis, artefatos de entrada e saída, assim como métricas a serem coletadas para monitoramento. Modelos de transformações também foram especificados para transformar os processos de experimentos especificados com as DSLs em *workflows* para os participantes do experimento. Os *workflows* gerados são instanciados e executados por um sistema web que é alvo de outra dissertação.

Apesar do método proposto neste trabalho poder ser aplicado a outras abordagens generativas, os resultados do estudo exploratório realizado, além de avaliá-lo, também contribuíram para a definição e melhoria da abordagem generativa de experimentos controlados proposta em (FREIRE, ACCIOLY, *et al.*, 2013),

[1] Integram o grupo de pesquisa cinco pesquisadores da Universidade Federal do Rio Grande do Norte, sendo uma aluna de doutorado, dois alunos de mestrado e dois professores doutores do Programa de Pós-graduação em Sistemas e Computação. Além disso, colaboram com o projeto pesquisadores da Universidade Federal de Pernambuco.

focando especificamente nos problemas de composição de DSLs. Os problemas de composição de DSLs já vinham sendo investigados pelo grupo e foram relatados em (CAMPOS NETO, BEZERRA, *et al.*, 2012; CAMPOS NETO, FREIRE, *et al.*, 2013).

1.5. Organização do Documento

O restante deste documento está estruturado da seguinte forma.

O *capitulo 2* apresenta os principais referenciais teóricos que serviram de base para a construção desta pesquisa, trazendo uma visão geral sobre engenharia de LPS, desenvolvimento generativo, composição de DSLs e integração entre FMs e DSLs.

O *capítulo 3* apresenta o método proposto para desenvolvimento com múltiplas DSLs em abordagens generativas, primeiramente apresentando uma visão geral do método e em seguida detalhando cada uma das fases e atividades que o compõem.

O *capítulo 4* descreve o estudo exploratório que foi realizado para avaliar o método proposto, apresentando os artefatos gerados e discutindo os resultados da aplicação da abordagem em um cenário real.

O *capítulo 5* apresenta os trabalhos relacionados com a nossa abordagem e uma análise comparativa entre algumas abordagens da literatura e a nossa.

O *capítulo 6* apresenta as considerações finais, principais contribuições e limitações do trabalho, bem como trabalhos futuros de pesquisa que podem ser desenvolvidos como continuidade desta dissertação.

2. FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta uma visão geral das bases teóricas que fundamentam as proposições e discussões contidas nesta dissertação, tais como Engenharia de Linha de Produto (Seção 2.1), com ênfase na definição de Abordagens Generativas; Composição de Linguagens Específicas de Domínio (Seção 2.2), destacando os problemas envolvidos e soluções específicas; uma Abordagem Proposta para utilização de DSLs na Engenharia de Linha de Produto (Seção 2.3); e, por fim, uma visão geral das principais tecnologias utilizadas no desenvolvimento do trabalho.

2.1. Engenharia de Linha de Produto

O objetivo da Engenharia de Linha de Produto (*Product Line Engineering* – PLE) é gerenciar de modo eficiente um conjunto de produtos provenientes de uma Linha de Produto de Software (LPS) (VOELTER e VISSER, 2011). Uma LPS representa uma família de sistemas de software, que partilham um conjunto de funcionalidades comuns e que satisfazem as necessidades de um segmento de mercado particular, desenvolvidas de forma sistemática, a partir de um conjunto de artefatos-base que formam a arquitetura da LPS (CLEMENTS e NORTHROP, 2011). O conceito de LPS está intrinsecamente ligado às áreas de arquitetura, engenharia e reuso de software e envolve a definição de variados processos de desenvolvimento de LPSs, com forte aceitação no meio acadêmico e industrial.

Segundo (POHL, BÖCKLE e LINDEN , 2005), o desenvolvimento de uma LPS pode ser definido basicamente em duas fases: Engenharia de Domínio (ED), que envolve a análise de domínio da aplicação, a delimitação do escopo da LPS e o desenvolvimento dos artefatos reusáveis que serão utilizados para definir uma arquitetura comum para a LPS; e Engenharia de Aplicação (EA), responsável pela geração dos produtos da LPS, através do reuso dos artefatos gerados durante a ED. Uma LPS especifica um conjunto de produtos de software que contém características genéricas similares que permite a definição de uma infraestrutura comum de estruturação de artefatos (CZARNECKI e EISENECKER, 2000).

Em um processo típico de desenvolvimento de LPS (CZARNECKI e EISENECKER, 2000), após a especificação e implementação da arquitetura de LPS

na ED, ocorre a etapa de EA. Durante esse estágio, os artefatos de implementação produzidos anteriormente são compostos e integrados para gerar uma instância (produto) da LPS. Esse processo é também conhecido como derivação de produto (DEELSTRA, SINNEMA e BOSCH, 2005). Características (*features*) da LPS são usadas para capturar pontos em comuns e discriminar variabilidades entre os produtos de uma LPS. Uma *feature* pode representar uma propriedade do sistema ou uma funcionalidade relevante para o seu interessado, também chamado de *stakeholder*. A derivação do produto pode ser também entendida como o processo de especificação de um produto a partir de uma configuração de artefatos que implementam uma seleção de *features* da LPS. A Figura 1 apresenta uma visão geral da geração de produtos a partir dos ativos-base de uma LPS. O processo de geração de produtos necessariamente recebe como entradas os ativos-base desenvolvidos, e uma configuração de produto que consiste na escolha das *features* opcionais e alternativas, que irão compor o produto junto aos ativos comuns a todos os produtos. O mecanismo de geração recebe estas entradas e faz a composição gerando como saída os produtos da LPS (KRUEGER, 2006). O escopo da LPS pode ser determinado através dos possíveis conjuntos de produtos gerados a partir destas entradas.

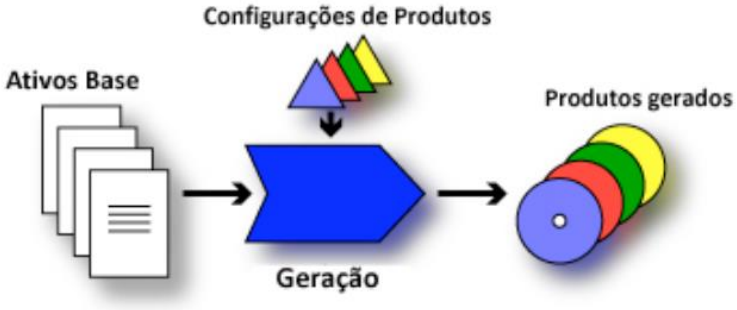


Figura 1. Modelo geral do processo de geração de produtos em uma LPS – Adaptado de *(KRUEGER, 2006)*

A utilização de uma ferramenta de apoio que facilite a seleção, composição e configuração dos artefatos e suas respectivas variabilidades é o ideal para a realização desta prática. Abordagens modernas de engenharia de software, tal como o Desenvolvimento Generativo (CZARNECKI e EISENECKER, 2000), oferecem fundamentos para a definição de técnicas e mecanismos que auxiliam na automação do processo de derivação de produtos. O objetivo central dessas abordagens é propiciar melhorias na qualidade do processo de derivação. DSLs e geradores de código são duas das principais tecnologias adotadas com tal propósito.

2.1.1. Modelo de Feature

Um modelo de *feature* (*Feature Model* – FM) é uma representação compacta das características (*features*) dos possíveis produtos de uma LPS, assim como suas restrições (VOELTER e VISSER, 2011). Ele apresenta informações sobre as *features* comuns e variáveis de uma linha de produto em seus diferentes níveis de abstração, constituindo uma representação hierárquica de um conjunto de *features* com diferentes relacionamentos entre si. Um FM modela todos os possíveis produtos de uma LPS em um dado contexto, representando assim uma família de produtos, e não um único produto, como em outras representações de modelos.

A primeira formulação de uma linguagem para modelar *features* foi proposta por (KANG, COHEN, *et al.*, 1990) e deu origem a abordagem nomeada como *Feature-Oriented Domain Analysis* (FODA), que acabou consolidando-se e tornando-se, hoje, uma das estratégias mais adotadas para gerenciamento de *features*. A abordagem FODA prevê a criação de um modelo de *features* geralmente organizado em uma estrutura de árvore, na qual ao selecionar uma *feature*-folha, todas as *features* acima desta até a raiz, também devem ser selecionadas. Neste modelo, as *features* podem ser classificadas como: (i) obrigatórias — quando representam artefatos comuns a todos os produtos da LPS, que devem sempre ser incluídos em todos os produtos gerados; (ii) opcionais — quando estão associados a artefatos cuja inclusão depende da seleção da *feature*, sendo facultada a escolha de selecioná-la ou não ao engenheiro de aplicação; e (iii) alternativas — quando são constituídas de agrupamentos de duas ou mais *features* mutuamente exclusivas; e (iv) ou-*feature* — quando agrupamento de *features* não mutuamente exclusivas.

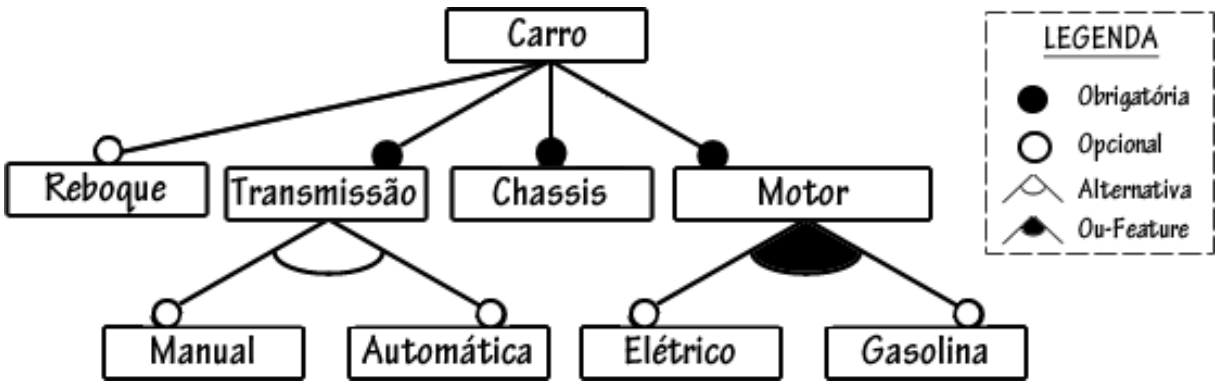


Figura 2. FM de uma LPS de domínio automotivo – Adaptado de (CZARNECKI e HELSEN, 2006)

A Figura 2 ilustra um FM apresentado em (CZARNECKI e EISENECKER, 2000) para representar *features* comuns e variáveis existentes em uma LPS voltada para um domínio automotivo. No exemplo em questão, temos a *feature* raiz (Carro) que possui quatro *features* relacionadas, sendo três dessas obrigatórias (Transmissão, *Chassis* e Motor) e outra opcional (Reboque). A *feature* Transmissão, por sua vez, é uma *feature* alternativa com duas opções mutuamente exclusivas (Manual ou Automática), enquanto a *feature* Motor é uma ou-*feature* com duas opções variações (Elétrico ou Gasolina), não mutuamente exclusivas.

2.1.2. Abordagem de Desenvolvimento Generativo

Esta seção apresenta uma visão geral de uma abordagem generativa que inclui DSLs na Engenharia de Domínio e Aplicação e foi proposto por (CZARNECKI e EISENECKER, 2000) (Figura 3). Seus proponentes definem o desenvolvimento de software generativo como sendo uma abordagem para famílias de sistemas, focada na automação e criação membros destas famílias: a partir de uma ou mais especificação textual ou gráfica que utilize modelos de *features* ou DSLs, um dado sistema pode ser automaticamente gerado da família.

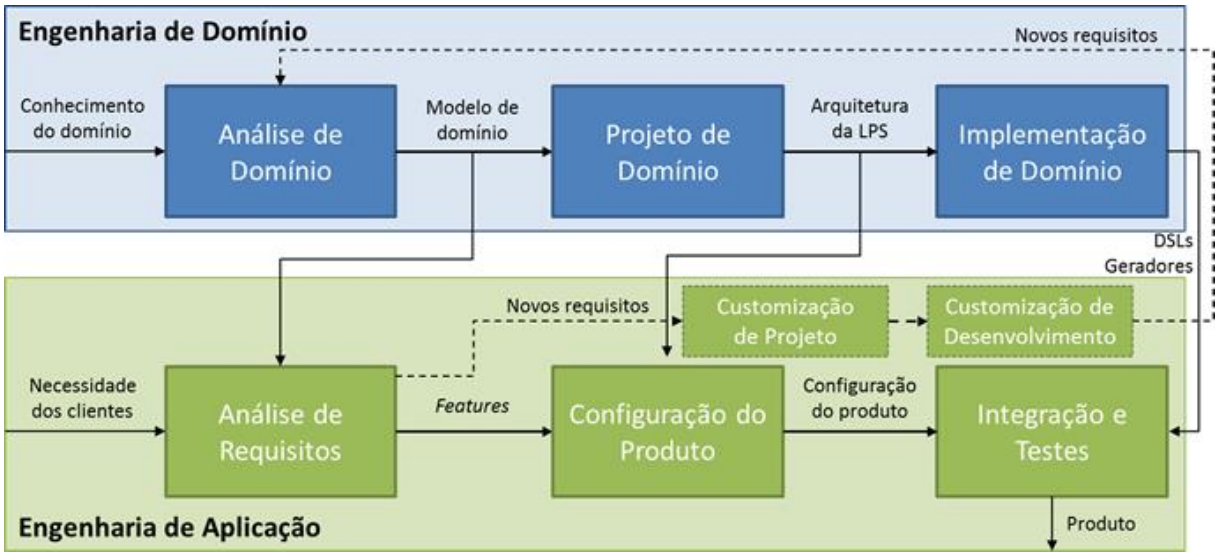


Figura 3. Visão Geral da Abordagem Generativa do (CZARNECKI e EISENECKER, 2000)

A Figura 3 apresenta um resumo esquemático das etapas da abordagem, dividida em: (i) Engenharia de Domínio, focada na identificação dos requisitos variáveis e comuns para consequente definição da arquitetura da LPS com DSLs e geradores de código; e (ii) Engenharia de Aplicação, que contém passos destinados

à configuração de um produto específico e sua derivação. É possível observar ainda na Figura 3 os artefatos fornecidos como entrada e gerados como saída em cada etapa da abordagem, além dos caminhos alternativos que levam a customização do projeto e desenvolvimento.

Cada uma das etapas está dividida ainda em fases menores. Na Engenharia de Domínio são executadas atividades que visam, sobretudo, a produção dos artefatos necessários para implementação da LPS. Essas atividades estão divididas nas fases de análise, projeto e implementação. Na primeira delas, na de análise, a partir de artefatos que contemplem o conhecimento do domínio, é feito o levantamento dos requisitos do domínio, normalmente especificados através de modelos de *features*, gerado como saída desta fase e entrada da seguinte. Na fase de projeto, modelos de análise são refinados visando à elaboração de uma arquitetura do domínio, também chamada de Arquitetura de Software Específica para Domínios (*Domain-Specific Software Architecture* – DSSA), a arquitetura da LPS. Por último, a fase de implementação visa à geração de código fonte para a arquitetura gerada no projeto do domínio. Nesta também surgem as linguagens específicas de domínios e geradores da LPS, que são em geral responsáveis pela especificação e geração do código de variabilidades da arquitetura da LPS. Por sua vez, a Engenharia de Aplicação, onde o objetivo é desenvolver produtos/aplicações a partir dos artefatos produzidos na Engenharia de Domínio, está dividida nas fases de análise de requisitos, configuração do produto e integração e testes. Na fase de análise de requisitos o objetivo é, a partir das necessidades dos clientes, levantar requisitos para definir o escopo do produto que se pretende gerar, resultando numa seleção de *features* que implemente os requisitos demandados. Essas *features* serão fornecidas como entrada para a fase seguinte, onde serão formalmente isolados os artefatos que os implementa, a partir de uma especificação de *features* geralmente formada por instâncias do modelo de *features* e em alguns casos modelos de DSLs. Essa especificação será usada como entrada dos geradores automatizados na fase de integração e testes para derivação do produto final desejado. É possível ainda que durante a fase de análise de requisitos novos requisitos inexistentes na LPS sejam identificados, sendo preciso retornar às etapas da engenharia de domínio para customização.

2.2. Composição de Linguagens Específicas de Domínio

Uma linguagem específica de domínio (*Domain-Specific Language* – DSL) é uma linguagem cuja expressividade é em particular problema do domínio, diferente das linguagens de propósito gerais, tais como Java e C++, que são projetadas para se apropriar de diferentes tipos de aplicações e cenários (CZARNECKI e EISENECKER, 2000). A definição dos conceitos de um domínio é realizada através da especificação de uma DSL, que pode ser textual, gráfica ou híbrida. Uma DSL serve ao propósito de tornar os elementos de um domínio formalmente expressáveis e modeláveis. Toda DSL possui uma gramática concreta, também conhecida como gramática BNF, que representa a sua sintaxe. Além disso, algumas delas possuem ainda uma gramática abstrata, representada através de um meta-modelo. A semântica de uma DSL deve ser bem documentada ou ser intuitivamente clara para o modelador, neste sentido, a DSL deve adotar conceitos do espaço do problema, de forma que o especialista do domínio reconheça sua “linguagem de domínio” (STAHL e VOLTER, 2005; FOWLER e PARSONS, 2013).

O desenvolvimento de sistemas de software utilizando linguagens específicas de domínio também vem aumentado nos últimos anos. A razão para isso é que as DSLs elevam o nível de abstração e trazem facilidades para geração de modelos ou código de níveis inferiores, trazendo assim o potencial de aumentar a produtividade do desenvolvimento de sistemas em diferentes domínios (JEDLITSCHKA, CIOLKOWSKI, *et al.*, 2007).

2.2.1. Problemas de Composição

O desenvolvimento de um sistema único composto de múltiplas DSLs tem apresentado novos desafios para as abordagens atuais de criação e manutenção de DSLs, que costumam tratar apenas de DSLs simples. Existem diversas questões que foram levantadas e vêm sendo tema de estudos (HESSELLUND e LOCHMANN, 2009), dentre as quais: qual o suporte dado por ferramentas e métodos para o desenvolvimento com múltiplas DSLs em um sistema único? Como os desenvolvedores podem ser auxiliados na navegação e manutenção da consistência entre programas/modelos escritos em DSLs diferentes? Como podemos oferecer

orientações como assistentes inteligentes com recursos de auto complemento para várias DSLs?

Existem alguns tipos de restrições que precisam ser mantidas durante o desenvolvimento de aplicações com múltiplas DSLs. Hessellund *et al.* realizaram um estudo de caso (HESSELLUND, CZARNECKI e WASOWSKI, 2007) para investigar essas restrições. No estudo, foram identificados quatro tipos de restrições que consideramos em nosso trabalho:

- *Artefatos individuais bem-definidos*: essa restrição é aplicável para os casos em que a presença de um elemento da linguagem exige a presença de outro elemento no mesmo contexto. Em algumas DSLs, a gramática da linguagem não é suficiente para realizar essa verificação, como é o caso de DSLs baseadas em XML, usadas no estudo de caso, por limitações do próprio meta-modelo. Um outro exemplo, na linguagem SQL, ao modelar uma consulta a dados que faça uso de uma função agregadora, como o "*count*", e que, em conjunto, seja especificado ao menos um campo para agregação no "*select*", a sintaxe da linguagem impõe a inclusão também de uma cláusula "*group by*" ao final da consulta para a modelagem se torne bem-definida;
- *Integridade entre artefatos*: no desenvolvimento com múltiplas DSLs, o modelo de uma DSL pode fazer referência a modelos de outras DSLs. São necessários mecanismos para garantir que uma mudança em uma DSL seja corretamente aplicada nas demais;
- *Referências com restrições adicionais*: um caso mais complexo de restrição ocorre quando o tipo de algum atributo em um modelo vai definir o tipo de outro atributo em outro modelo. O exemplo apresentado é o caso em que um campo de senha no modelo de entidades exige que a caixa de texto no modelo de formulários seja do tipo *password*, que oculta os caracteres;
- *Restrições de estilo*: restrições também podem dizer respeito a estilos de codificação, que não causa erros, mas proporcionam uma maior uniformidade entre os artefatos. Por exemplo, é comum que os nomes de classes sejam iniciados com letra maiúscula.

A gerência da consistência entre modelos é um dos principais desafios para o desenvolvimento com várias DSLs. Em (LOCHMANN e BRÄUER, 2007), é

apresentado um resumo das abordagens propostas na literatura para o problema dessa integração. Entre elas “referências baseadas em nome”, que utilizamos nesse trabalho. Essa abordagem conecta modelos referenciando elementos e outros modelos através de *Strings* com os nomes destes modelos. Com essa abordagem a semântica da relação entre os modelos é feita somente em nível de código, ficando fora do modelo em si. Esse problema poderia ser resolvido com referências tipadas.

Para (BÉZIVIN e JOUAULT, 2005), violações de restrições podem ainda ser classificadas de acordo com o nível de severidade entre *errors* e *warnings*. *Errors* são violações graves, que tornam os modelos inválidos, enquanto que *warnings* indicam algum problema, mas não invalidam o modelo. Outros níveis também podem ser definidos, melhorando a precisão das violações.

2.2.2. Método para Composição de DSLs

Para tentar resolver os problemas listados na subseção anterior, Hessellund *et al.* (HESSELLUND e LOCHMANN, 2009; HESSELLUND, 2009) propuseram a adoção de uma abordagem sistemática para promover o desenvolvimento com múltiplas DSLs. O método proposto por eles divide o desenvolvimento em três etapas: (i) Identificação; (ii) Especificação; e (iii) Aplicação, que são apresentadas a seguir.

1) Identificação: Esse é o primeiro passo do método e seu objetivo é descobrir pontos de sobreposições entre diferentes DSLs (HESSELLUND e LOCHMANN, 2009). Uma DSL se sobrepõe a outra(s) quando há uma referência entre elas. Segundo o método, há três possíveis tipos de referências, comentadas a seguir: (i) explícita, quando um elemento de um meta-modelo referencia, diretamente, um elemento em outro meta-modelo (HESSELLUND, CZARNECKI e WASOWSKI, 2007); (ii) implícita, quando não é possível expressar uma ligação direta entre os meta-modelos e a conexão é feita, por exemplo, por meio de valores *String* que equivalem a atributos em outros modelos; e (iii) semântica, geralmente associada a conexões mais complexas, onde a intenção semântica também precisa ser considerada para a especificação dos meta-modelos. Classificar referências deve ser tarefa dos especialistas de domínio, realizada de forma manual ou automática, e

de fundamental importância para a abordagem. É de acordo com o tipo de referência, que o especialista em composição, no passo seguinte, escolherá que implementação realizar para codificar as conexões entre os meta-modelos.

2) Especificação: A proposta deste passo é implementar as conexões identificadas no passo anterior, de acordo com o tipo de referência. A especificação pode ser: (i) parcial – quando só as conexões são codificadas; ou (ii) completa – quando todo o meta-modelo é representado na linguagem de conexão. Para a execução deste passo se faz necessária a adoção de alguma ferramenta de suporte, como a *SmartEMF*[2] (HESSELLUND, 2007), proposta em conjunto ao método.

3) Aplicação: O último passo, a aplicação, é onde as codificações realizadas no passo anterior são de fato utilizadas. É a aplicação que tornará visível os ganhos da adoção do método em questão, por meio de ações descritas em três áreas: (i) navegação; (ii) verificação de consistência; e (iii) apresentação de orientações (guias). A navegação é simplesmente uma forma de navegar entre os modelos; a verificação de consistência é a mais representativa das ações, pois facilita a busca por violações de restrições à consistência; e guias, é um conceito já bastante explorado pelas IDEs de desenvolvimento para apresentação de sugestões, *errors* e *warnings* aproveitados na metodologia em questão.

2.3. Engenharia de Linha de Produto utilizando DSLs

Em (VOELTER e VISSER, 2011) é discutido os limites de expressividade dos modelos de *features* e proposto como solução a integração de DSLs à Engenharia de Linha de Produto. Eles estudaram as diferenças de expressividade entre os modelos de *features* e DSLs e discutiram mecanismos de integração entre os dois, destacando que a escolha entre uma ou outra solução não é mutuamente exclusiva. O estudo mostrou que DSLs podem ser mais expressivas que os FMs por possibilitarem representar atributos, condições de recursividade, referências com restrições, entre outros recursos não suportados pelos modelos de *features*.

---

[2] A ferramenta estende o *plugin* Eclipse Modeling Framework (EMF) para facilitar o desenvolvimento com múltiplas DSLs baseadas em EMF. Por essa razão, recebe o nome “S*marter*” (mais inteligente) + *EMF*, resultando em *SmartEMF*.

A necessidade de atributos ocorre pela necessidade de expressar algumas características das *features* e assim maximizar o poder de configuração dessas. A recursividade possibilita a modelagem de repetições e expressões aninhadas que, apesar de possível de ser expresso em alguns tipos de modelos de *features*, tais como os baseados em cardinalidade, não se é possível especificar condições e limites de acordo com um contexto de configuração de *features*. Por fim, o uso de referências permite criar relações de contexto, mas, de modo similar ao anterior, maior parte dos modelos de *features* não possibilitam expressar regras de ligações entre tais, como em casos de referências com restrições.

2.4. Frameworks e Linguagens utilizadas

Esta seção apresenta um resumo das principais tecnologias, incluindo *frameworks* e linguagens, utilizadas para aplicação do método no estudo exploratório realizado nesta dissertação. Desse modo, é apresentada uma visão geral do *Eclipse Modeling Framework* (EMF), que foi utilizado para especificação dos meta-modelos construídos durante o estudo e a linguagem *xText*, usada para implementação das gramáticas das DSLs. Além desses, é apresentado o *framework* para mapeamento de variabilidades entre modelos, *FeatureMapper,* e a linguagem QVT para especificação de transformações entre modelos.

2.4.1. Eclipse Modeling Framework

O *Eclipse Modeling Framework* (EMF) é um *framework* para criação de modelos do Eclipse. Construído em linguagem Java/XML é útil na geração de ferramentas e outras aplicações baseadas em modelos de classe simples (STEINBERG, BUDINSKY, *et al.*, 2008). Uma das vantagens da sua utilização é a rápida validação de modelos em códigos Java, eficiente e facilmente customizáveis. Ele pretende fornecer os benefícios da modelagem formal, porém com um custo de entrada menor. Além da geração de código, EMF provê a capacidade de salvar objetos como documentos XML para estabelecimento de comunicação ou troca de dados com outras ferramentas e aplicações (STEINBERG, BUDINSKY, *et al.*, 2008).

Um modelo é descrito no EMF usando conceitos que estão em um nível mais alto que classes e métodos simples. Para definir um modelo usando estes tipos de partes de modelo, nós precisamos de uma terminologia comum para descrever as partes, um modelo para descrever modelos EMF, chamado de *Ecore*. O *Ecore* é o modelo usado para representar modelos em EMF. É o que chamamos de modelo do modelo, ou meta-modelo. Quando instanciamos as classes definidas em *Ecore* para definir o modelo para nossa aplicação, nós estamos criando o que nós chamamos de modelo core. Um exemplo simplificado de um subconjunto do meta-modelo *Ecore* é mostrado na Figura 4.

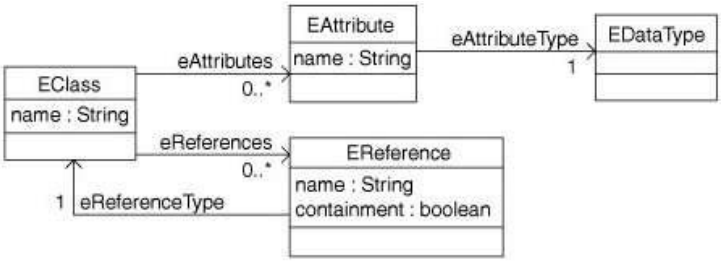


Figura 4. Exemplo simplificado de um subconjunto de elementos do meta-modelo *Ecore*

Este diagrama apresenta um resumo das partes de *Ecore* necessárias para criação de um modelo. Para construí-lo utilizamos a ferramenta EMF, *framework* do Eclipse próprio para criação de modelos. O EMF dispõe de visualizações do modelo em forma de árvore ou em forma de diagrama, ambos definidos na pasta *model* do projeto.

2.4.2. xText

O *xText* (http://eclipse.org/xtext) é um *framework* para desenvolvimento de linguagens textuais baseado no meta-modelo *Ecore* e implementado sobre o ambiente Eclipse (BETTINI, 2011). Com isso, o *xText* possibilita a criação de DSLs textuais completas aproveitando os mecanismos e funcionalidades existentes no ambiente de desenvolvimento Eclipse. O *xText* permite validar modelos especificados com as gramáticas desenvolvidas no seu ambiente a partir da especificação de códigos em linguagem Java. Além disso, é possível representar/extrair ou gerar uma gramática *xText* a partir de um meta-modelo *Ecore*.

2.4.3. FeatureMapper

O *FeatureMapper* (http://featuremapper.org) é uma abordagem ferramental que combina técnicas do desenvolvimento dirigido por modelos com a engenharia de linha de produtos (HEIDENREICH, KOPCSEK e WENDE, 2008). A ferramenta permite representar *features* no espaço problema usando um modelo features, mapeando as *features* do FM para soluções de artefatos expressos em linguagem EMF, baseados no meta-modelo *Ecore*, tal como DSLs construídas no *xText*. A ferramenta provê ainda visualizações específicas do mapeamento, permitindo realizar transformações a partir de seleções de *features* que gerem como saída modelos personalizados, conforme seleção das *features*.

A ferramenta *FeatureMapper* consiste de múltiplos plug-ins construídos para a plataforma Eclipse que prover uma extensível interface que suporta desde a construção de modelos de *features* específicos até a derivação de modelos a partir de uma configuração de *features*.

2.4.4. Linguagem de transformação de modelos: QVT

Transformações de modelo para modelo (M2M) é um dos principais alicerces do desenvolvimento dirigido por modelos (*model-driven development* – MDD). A especificação QVT faz parte de um subprojeto da comunidade Eclipse que dispõe de soluções para transformações entre modelos. As iniciais QVT derivam do conjunto de ações que fundamentam a sua atuação: *Query*, *View* e *Transformation* (OMG, 2011). A especificação QVT tem uma natureza híbrida declarativo-imperativa, com a parte declarativa (QVTd) sendo dividida em uma arquitetura de dois níveis, que funcionam como base para a semântica de execução da parte imperativa. A função da QVTd é prover a parte estrutural, uma IDE baseada no Eclipse para as linguagens *Core* QVT (QVTc) e *Relational* QVT (QVTr). A Figura 5 mostra os relacionamentos entre elementos da linguagem QVT.

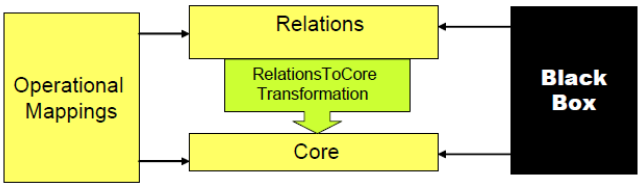


Figura 5. Relacionamentos entre elementos QVT (OMG, 2011)

Além das relações declarativas e linguagens de núcleo, existem dois mecanismos que trabalham com implementações imperativas de transformações de relações ou núcleo: uma língua padrão para mapeamentos operacional (*Operational QVT* – QVTo) e uma não padrão, chamada de caixa-preta, onde se pode plugar outras implementações (OMG, 2011). A QVTo contém instruções detalhadas de execução, ao contrário da QVTr que omite estes passos explícitos e conta com a correspondência entre os elementos individuais da entrada e saída de domínio (OMG, 2011). Consequentemente, códigos QVTo requerem um algoritmo completo, ou seja, como produzir um modelo de saída a partir do modelo de entrada inteiro, enquanto o QVTr contém rotinas para elemento para elemento de mapeamento.

A QVTo permite especificar transformações operacionais, operações de mapeamento, etc., a partir de modelos de entrada. Uma transformação operacional representa a definição de uma transformação unidirecional que se expressa imperativamente. Possui uma assinatura que indica os modelos envolvidos na transformação e define uma operação de entrada, chamada principal, que representa o código inicial a ser executado para realizar a transformação. Uma operação de mapeamento é uma implementação de um mapeamento entre um ou mais elementos de origem do modelo em um ou mais elementos do modelo alvo.

2.5. Sumário

Este capítulo apresentou as bases teóricas para o desenvolvimento desta dissertação. O método proposto neste trabalho baseia-se na integração de abordagens de desenvolvimento generativo existentes com um método focado no desenvolvimento com composição de múltiplas DSLs. Foram apresentadas também as principais tecnologias utilizadas durante o estudo exploratório que avaliou a aplicação do método proposto.

3. UM MÉTODO PARA DESENVOLVIMENTO DE ABORDAGENS GENERATIVAS COM COMPOSIÇÃO DE LINGUAGENS ESPECÍFICAS DE DOMÍNIO

Este capítulo apresenta um método de desenvolvimento de abordagens generativas com múltiplas linguagens específicas de domínio (*Domain-Specific Languages* – DSLs) proposto como uma das contribuições desta dissertação. Inicialmente, é apresentada uma visão geral da nossa abordagem (Seção 3.1). A utilização de DSLs no desenvolvimento generativo já é considerada por alguns pesquisadores (CZARNECKI e EISENECKER, 2000), assim como o seu uso como alternativa e complemento ao modelo de *features* (VOELTER e VISSER, 2011). Contudo, boa parte dos métodos de desenvolvimento de abordagens generativas já propostos não aborda de forma explícita a questão de desenvolvimento de múltiplas DSLs. Alguns trabalhos de pesquisa (HESSELLUND, 2009; LOCHMANN e HESSELLUND, 2009) discutiram soluções para o desenvolvimento com múltiplas DSLs, mas sem considerar as abordagens generativas. Dessa forma, este trabalho propõe um método para o desenvolvimento de abordagens generativas centrado na integração do modelo de *feature* (*feature model* – FM) com composição de DSLs. As etapas e atividades do método estão descritas em detalhes na Seção 3.2, assim como os artefatos recebidos e gerados a cada fase.

3.1. Visão Geral do Método

O desenvolvimento generativo de software objetiva especificar, modelar e implementar famílias de sistemas de modo que um dado sistema possa ser automaticamente gerado a partir de uma especificação de *features*, expressa, em geral, através de alguma DSL. Ao longo dos últimos anos, diversos métodos de desenvolvimento de abordagens generativas têm sido propostos pela comunidade (GREENFIELD, SHORT, *et al.*, 2004; GROHER, FIEGE, *et al.*, 2011; ZSCHALER, SÁNCHEZ, *et al.*, 2011). Nosso método adota como base a abordagem proposta por (CZARNECKI e EISENECKER, 2000) devido esta já prever a utilização de DSLs para especificar produtos a serem derivados. Tal abordagem é organizada nas duas principais fases existentes na engenharia de linhas de produtos (*Product Line Engineering* – PLE): (i) a Engenharia de Domínio – que focaliza a identificação dos

requisitos variáveis e comuns para consequente definição da arquitetura da LPS com seus artefatos reusáveis; e a (ii) Engenharia de Aplicação – que contempla atividades destinadas à configuração de um produto específico e sua derivação. Nosso método está focado nas fases de projeto e implementação de domínio e na derivação de produtos.

Voelter e Visser (2011) contrastam a aplicação de DSLs em PLE como meio termo entre a modelagem de *features* e os artefatos da LPS. Eles analisam os limites de expressividade dos modelos de *feature* (*feature models* – FMs) e mostram que DSLs podem ser usadas de forma complementar nesses casos. DSLs não representam uma alternativa mutualmente exclusiva aos FMs, elas podem ser utilizadas em conjunto aos FMs de forma a ampliar as possibilidades da derivação de produtos de uma LPS. Como também estamos interessados em investigar as diferentes estratégias para combinar modelos de *features* e DSLs, tanto na engenharia de domínio quanto de aplicação de uma LPS, a abordagem de Voelter e Visser também contribuiu para o desenvolvimento do nosso método.

No contexto de domínios complexos, a utilização de mais de uma DSL para representar diferentes visões/perspectivas da aplicação ou família de aplicação sendo desenvolvida tem se tornado cada vez mais comum, surgindo assim a necessidade de explorar soluções para implementar a composição entre elas (HESSELLUND, CZARNECKI e WASOWSKI, 2007). Essa questão é abordada por (HESSELLUND, 2009) ao propor um método que prevê etapas para identificar, especificar e aplicar a composição entre DSLs. Esse trabalho teve papel importante na definição do método de desenvolvimento de abordagens generativas proposto nesta dissertação, especificamente nas atividades de identificação, especificação e aplicação para composição de DSLs.

A Figura 6 apresenta uma visão geral do método proposto. A partir de uma entrada formada pelos requisitos do domínio e modelos de *features* e de arquitetura, o método prevê a identificação e implementação de múltiplas DSLs usadas para derivar produtos. A estrutura geral do método se baseia na abordagem generativa herdada, com foco nas etapas que envolvem o uso de DSLs (projeto/implementação de domínio e derivação). Foram incluídas atividade com funções específicas em cada etapa para suportar a criação e composição das DSLs como uma alternativa aos limites da expressividade do FM.

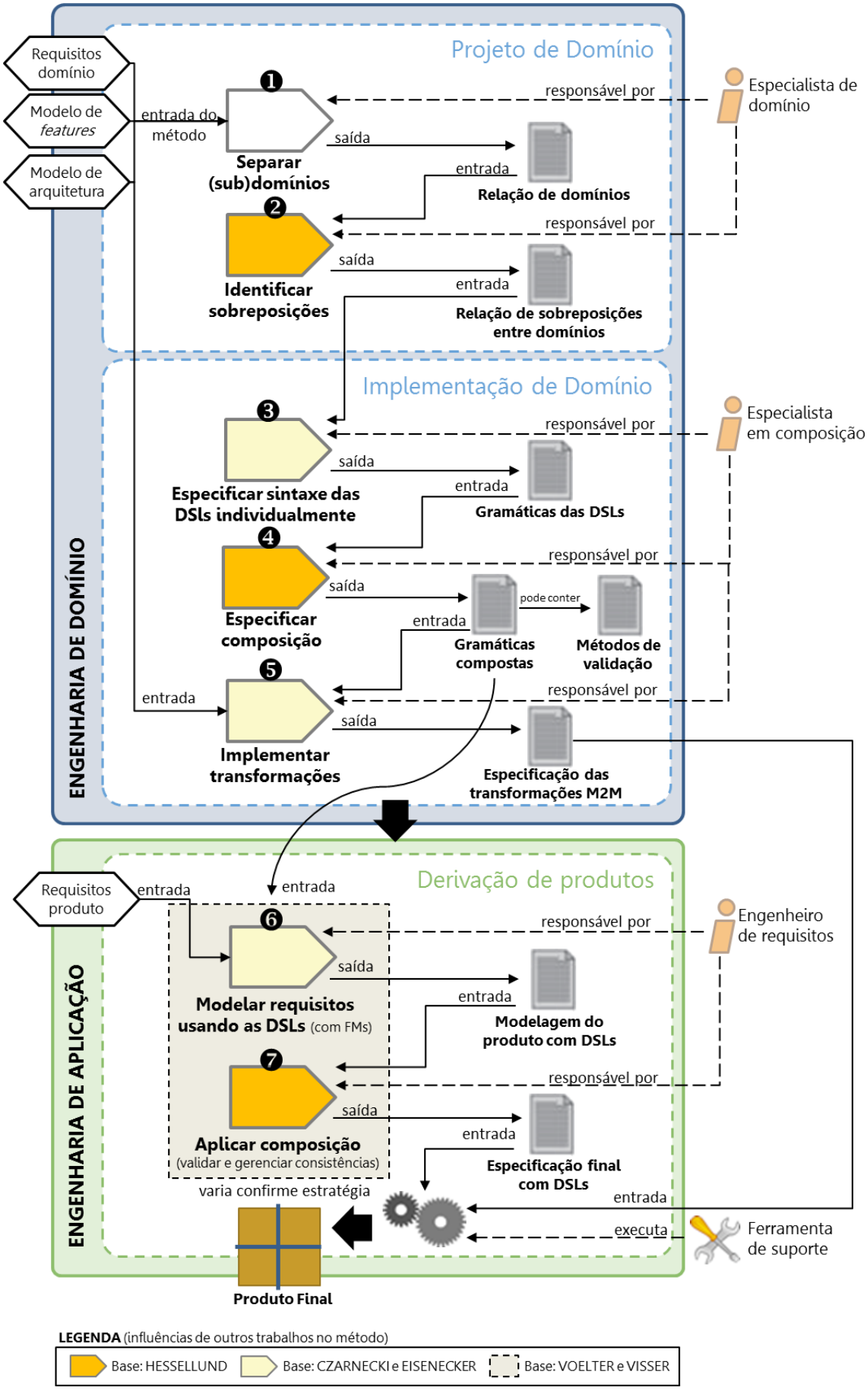


Figura 6. Visão Geral do Método

O método divide-se em três etapas: (i) Projeto de Domínio, (ii) Implementação de Domínio e (iii) Derivação de produtos, sendo as duas primeiras pertencentes à Engenharia de Domínio e a última à Engenharia de Aplicação. Cada etapa possui atividades/tarefas que são executadas por papéis específicos para gerar um conjunto de artefatos. A maioria das atividades herdam características das abordagens que influenciaram o método e estão representadas na legenda da Figura 6. O método apresenta ainda estratégias alternativas para derivação usando apenas DSLs ou combinando-as com o modelo de *features*. Mais detalhes da sua estrutura são apresentados na seção a seguir.

3.2. Estrutura do Método

Nosso método é composto por três etapas, as duas primeiras pertencentes a Engenharia de Domínio e a última referente à derivação de produtos na Engenharia de Aplicação. Cada etapa possui um conjunto de atividades que define os passos para identificação, criação e utilização de DSLs durante o desenvolvimento generativo. A etapa de derivação apresenta duas diferentes estratégias para geração de produtos da LPS: (i) a primeira usando apenas DSLs; e (ii) a outra combinando DSLs com FMs. As seções a seguir apresentam detalhes da metodologia de cada uma das etapas.

3.2.1. Projeto de Domínio

O objetivo desta primeira etapa é identificar a existência de diferentes domínios e suas interações. A partir de um conjunto formado pela especificação dos requisitos da LPS, as respectivas *features* representadas em um FM e seu modelo de arquitetura, um especialista de domínio é responsável por analisar os modelos de entrada para identificar quais conjuntos de abstrações são mais fortemente conectados, de modo que possam caracterizar um domínio. A representação de um domínio deve obedecer a critérios bem definidos que justifiquem a separação dos conceitos. Este trabalho adota a definição dada por (CZARNECKI e EISENECKER, 2000) para o termo domínio, que o compreende como uma área de conhecimento ou atividades caracterizadas por um conjunto de conceitos e terminologias conhecidos

pelos praticantes daquela área. Alguns pesquisadores (SIMOS, CREPS, *et al.*, 1996) distinguem ainda a existência de: (i) domínios do mundo real – que encapsulam conceitos do mundo real, tais como os domínios bancários e comerciais; dos (ii) domínios de sistemas – quando simplesmente agrupam característica de configurações de um sistema que podem ser reutilizadas, tais como um conjunto de classes do paradigma de orientação a objetos. Desse modo, domínios podem surgir tanto pela necessidade de reuso, como pelas afinidades de abstrações do mundo real, ou mesmo para representar, em escopos amplos, partes menores, com visões e soluções específicas. Cada visão contempla um domínio específico, com aspectos e conceitos distintos, que necessariamente se interligam.

Assim sendo, a segunda tarefa do projetista de domínio é encontrar interligações entre os domínios identificados. As ligações se darão por meio de sobreposições, que ocorrerão sempre que um domínio se sobrepor a outro(s) por meio de uma referência entre eles. Essa referência muitas vezes é estabelecida pelos próprios componentes daquele domínio. A quebra do domínio em partes específicas irá repercutir no FM, que passará a ter suas variabilidades mapeadas em diferentes visões, com soluções técnicas mais aprimoradas para resolver os problemas daquele aspecto em questão. É nesse contexto que surge a necessidade do desenvolvimento de múltiplas DSLs, as quais são planejadas para atender problemas específicos de cada domínio, e na necessidade de definição da utilização de mecanismos para projetar a composição das DSLs.

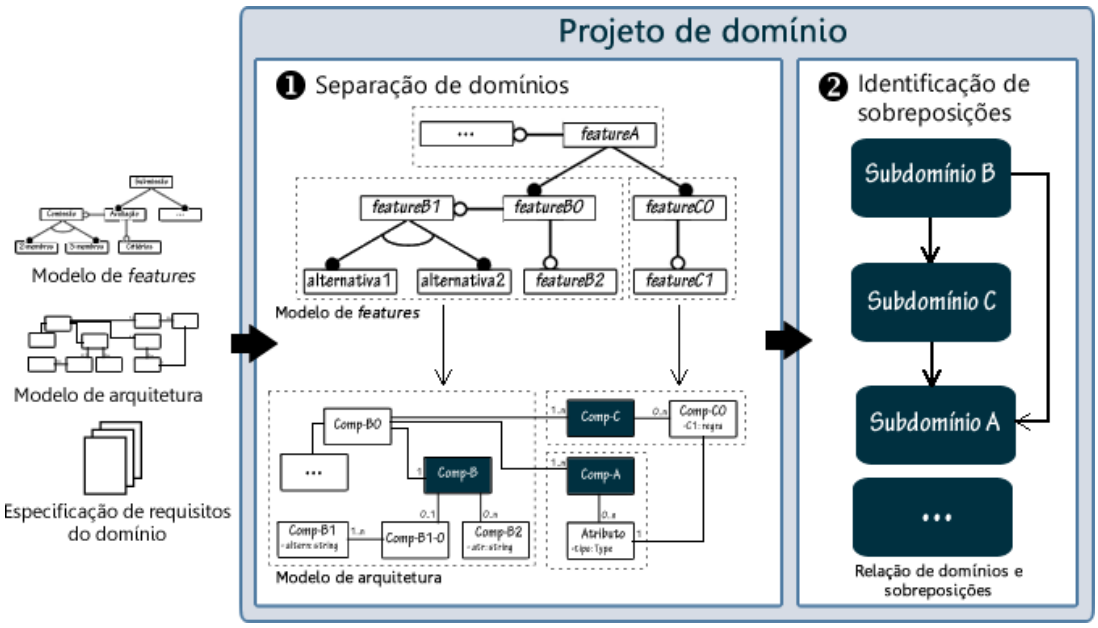


Figura 7. Exemplificação do fluxo de atividades da etapa de Projeto de Domínio

A Figura 7 exemplifica o fluxo de atividades a ser seguido na etapa de Projeto de Domínio. Esta etapa recebe como entrada um modelo de *features*, um modelo de arquitetura e uma especificação dos requisitos do domínio da LPS (ou família de sistemas) a ser desenvolvida. Os artefatos de entrada então devem ser analisados por um especialista do domínio para cumprimento de duas atividades. Na primeira (Figura 7 – Atividade 1), o objetivo é identificar afinidades entre os elementos arquiteturais, *features* e requisitos da LPS de modo a agrupá-los em subdomínios. Cada subdomínio deve representar uma visão ou perspectiva diferente do domínio, que podem corresponder a domínios do mundo real ou de sistemas. Em seguida, a segunda atividade (Figura 7 – Atividade 2) prevê a identificação de referências entre os subdomínios, resultando numa relação de domínios e sobreposições existentes.

Em resumo, são previstas para o Projeto de Domínio as seguintes atividades:

- *Separação de domínios*: Identificar e separar os domínios existentes no escopo de uma LPS (ou família de sistemas), tomando como base os modelos de entrada e adotando critérios pré-definidos que justifiquem a organização dos domínios em subdomínios. O resultado dessa atividade resultará em partes que representam aspectos específicos do cenário, possível de serem reusadas e que necessitam de soluções de implementação específicas;
- *Identificação de sobreposições*: Após a organização do domínio em subdomínios, é necessário verificar a ocorrência de sobreposições entre eles e classificar essas sobreposições quanto ao grau de entrelaçamento dos subdomínios.

A identificação de sobreposições implicará na implementação da composição de DSLs. Depois de identificadas, as referências são classificadas quanto à complexidade e natureza do entrelaçamento dos modelos. A tarefa de classificação está contida na atividade de identificação das sobreposições e é de fundamental importância para a abordagem, pois é de acordo com o tipo de referência, que o especialista em composição, na fase seguinte, escolherá qual implementação realizar para codificar as conexões entre os (sub)domínios. A existência de referências implicará em alguns tipos de restrições que precisam ser mantidas durante o desenvolvimento de aplicações com múltiplas DSLs.

3.2.2. Implementação de Domínio

A etapa de implementação de domínio envolve a escolha das tecnologias que serão usadas para implementar as linguagens específicas de domínio. Seu objetivo é implementar soluções específicas para os domínios projetados de forma a atender as variabilidades da LPS. Como resultado desta fase, temos a implementação dos artefatos de código reusáveis (*code assets*) da LPS, assim como o desenvolvimento e composição das gramáticas das DSLs necessárias para a derivação automática de produtos da LPS. A criação de linguagens específicas de domínio, sejam textuais ou gráficas, surge pela necessidade de ampliar a expressividade de abstrações de domínio, limitada pela representação do modelo de *features* (VOELTER e VISSER, 2011). De forma a exemplificar, apresentamos a seguir três recursos próprios das gramáticas que não têm adequado suporte do modelo de *features*:

(i) *Atributos* ─ expressam características das *features* e são usados para maximizar o poder de configuração da *feature*, quando selecionada. Apenas algumas ferramentas de FMs fornecem esse suporte;

(ii) *Recursividade* ─ possibilita a modelagem de repetições e expressões aninhadas que, apesar de poderem ser expressos em FMs baseados em cardinalidade, não permite a especificação de condições e limites de acordo com um contexto de configuração de *features*;

(iii) *Referências* ─ permite criar relações de contexto, mas, de modo similar à recursividade, modelos de *features* não possibilitam expressar regras de ligações entre tais, como em casos de referências com restrições.

A inclusão desses elementos no modelo de *features*, mesmo quando possíveis, geram um aumento na complexidade do modelo, dificultando o seu entendimento (VOELTER e VISSER, 2011). Nesse caso, podendo limitar a derivação de produtos de uma LPS. Tal fato ocorre pelas características limitadas dos FMs de expressar apenas variabilidades com números fixos e predeterminadas, o que impediria a aplicação de tais alterações. Ao invés disso, a criação de DSLs traz soluções mais viáveis. Em alguns casos, devido a tais limites de expressividade do modelo de *features*, é necessário o uso de outros tipos de DSLs (grafos, linguagens com recursividade, etc.) com representações mais expressivas para os usuários finais (*wizard*, arquivos de configurações, modelo, etc.).

Neste contexto, a separação do domínio em subdomínios é importante para que possamos pensar em soluções específicas para cada uma delas, implementadas por meio do desenvolvimento de DSLs. Contudo, além de especificar a sintaxe de cada DSL individualmente, é tarefa desta etapa especificar os pontos de sobreposições entre elas, identificados na etapa anterior, e implementar transformações de modelos, quando necessárias, para a geração final dos artefatos da LPS. A definição da sintaxe de uma DSL é realizada através de uma gramática BNF que especifica os elementos previstos no domínio. A forma de especificar a sintaxe depende da tecnologia escolhida. Essa escolha deve ter como base o conhecimento técnico da equipe e a capacidade da tecnologia de implementar as necessidades do domínio. O formalismo das DSLs é capaz de expressar as características limitadas no modelo de *features*.

Desse modo, as atividades previstas para etapa são:

- *Especificação da sintaxe das DSLs individualmente*: é necessário escolher a tecnologia a ser utilizada e implementar uma gramática BNF que contemple as variabilidades da LPS em cada domínio. Devem-se tomar como base os modelos de *features* e arquitetura separados por domínios, bem como a especificação de requisitos. A sintaxe da DSL deve permitir especificar qualquer produto a serem gerados da LPS, inclusive aqueles que não são eventualmente possíveis usando o FM;
- *Especificação da Composição das DSLs*: o segundo passo é analisar a relação de domínios e sobreposições resultante da etapa de projeto para codificar nas gramáticas as referências existentes. A codificação dependerá do grau de complexidade das referências e da tecnologia usadas para a criação das gramáticas. Essa atividade resultará em mudanças nas gramáticas construídas anteriormente e, quando necessário, implementação de métodos adicionais à gramática para validar algumas restrições;
- *Implementação das transformações*: por fim, é necessário implementar transformações que possibilitem definir o mapeamento entre um produto modelado usando as DSLs criadas e que gerem um produto implementado em termos dos artefatos de código reusáveis definidos para a LPS. Essa atividade resultará em um artefato de código com a especificação das transformações prontas para ser executada nas etapas seguintes.

Uma tarefa complementar para a atividade de especificação da composição é a implementação de restrições, inerentes aos novos desafios oriundos da composição de DSLs. Restrições devem ser especificadas para validar, principalmente, a consistência entre modelos. Deve-se sempre que possível utilizar ferramentas que forneçam suporte à modelagem com sugestões de auto complemento, recursos de navegação entre os elementos e validação dos modelos. Em alguns casos, tais restrições precisam ser implementadas de forma a complementar as regras definidas pela gramática da DSL.

3.2.3. Derivação de Produtos

A derivação de produtos é a etapa final do método e está inserida no contexto da Engenharia de Aplicação. O objetivo dessa etapa é desenvolver produtos/aplicações a partir dos artefatos produzidos na Engenharia de Domínio. A geração de novos produtos pode ocorrer de forma automatizada, especialmente no contexto de definição de abordagens generativas com suporte de DSLs. O resultado da derivação pode ser um produto final parcial ou completo, de acordo com a estratégia de derivação utilizada. Nosso método prevê a derivação utilizando apenas DSLs (Seção 3.2.3.1) ou combinando DSLs e FMs (Seção 3.2.3.2). Não há confronto entre as estratégias, elas representam alternativas distintas com fins distintos para o mesmo objetivo.

3.2.3.1. Estratégia de derivação usando apenas DSLs

Nessa estratégia, as DSLs desenvolvidas são utilizadas para modelar os produtos (sistemas) finais que se deseja gerar. Como estamos lidando com uma abordagem de composição, cada DSL é utilizada para especificar uma parte específica do produto final. Neste caso, é necessário ainda gerenciar a consistência entre os modelos e validar as restrições existentes. Sugestões de ajustes e alertas de erros são recursos complementares, porém importantes, que podem existir na ferramenta usada para modelar os produtos. Validações só são possíveis devido à implementação de restrições atreladas às gramáticas das DSLs no passo anterior. Tal suporte não é fornecido pelo modelo de *features*, que não possui tal

expressividade. No caso dessa estratégia, o modelo de *features* é usado para fins de rastreabilidade dos elementos da LPS – mapeamento dos requisitos/*features* com elementos definidos na gramática das DSLs. Tal informação é essencial para manutenção e evolução da LPS. A seleção de fato das características que definem o produto a ser derivado deve ser realizada a partir de modelagens em cada uma das diferentes DSLs. As transformações de modelo criadas na etapa de implementação de domínio serão utilizadas para, a partir da modelagem final do produto, gerar o produto final derivado da LPS.

A Figura 8 apresenta um resumo esquemático do fluxo de atividades de uma derivação utilizando apenas DSLs. Diferente dos métodos de Engenharia de Aplicação baseado em *features*, tal como FODA (KANG, COHEN, *et al.*, 1990), onde a configuração de um produto geralmente é definida por uma instância do modelo de *features* gerada a partir da escolha das *features* que estão presentes no produto desejado. No caso dessa estratégia, tal especificação ocorre através das especificações de DSLs que então definem as características do produto a ser derivado. Para isso, a derivação deve receber como entrada uma listagem com os requisitos desejados para o produto e as DSLs desenvolvidas na etapa anterior. A utilização de DSLs amplia a expressividade da abordagem em relação aos FMs e possibilita a representação de variabilidades como modelo (Figura 8 – Atividade 1), uma vez que a implementação da gramática de uma DSL prevê todos os elementos comuns e variáveis que uma modelagem deve possuir. Isso possibilita a validação dos modelos antes de gerar a especificação final do produto (Figura 8 – Atividade 2), garantindo que o produto especificado seja possível de ser gerado e não viole nenhuma restrição da LPS, podendo ser transformado para gerar o produto final, resultado da derivação.

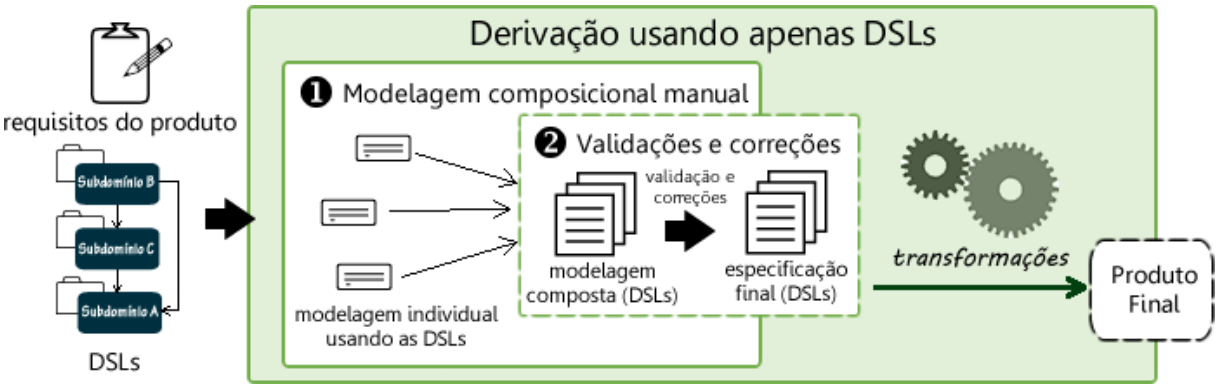


Figura 8. Fluxo de atividades na derivação usando apenas DSLs

Os códigos das transformações foram especificados ainda durante a fase de implementação, durante a atividade “Implementar transformações”. Esses códigos são executados agora, ao final da derivação, para gerar os artefatos de códigos do produto a ser derivado da LPS, devendo receber como entrada a especificação final do produto modelado usando as DSLs. É possível que mais de uma transformação sejam realizadas em sequência ou em paralelo, para ler informações provenientes das diferentes DSLs e processá-las sobre um único modelo. A forma como se dará a estratégia de transformação dependerá da linguagem de transformação de modelos escolhida durante a aplicação do método para codificar as transformações.

3.2.3.2. Estratégia de derivação combinado DSLs e FMs

Uma alternativa para derivar produtos no nosso método é combinando DSLs com um modelo de *features*. Essa estratégia é organizada em duas atividades. Na primeira, um FM é construído para representar as alternativas de modelagens mais comuns de uma LPS com cada uma de suas *features* endereçando um fragmento específico de modelagem especificada usando as DSLs. Ao selecionar um conjunto de *features* no FM é gerada automaticamente uma modelagem prévia do produto, montado pela junção dos fragmentos apontados pelas *features* selecionadas. O resultado desta primeira atividade é uma modelagem parcial que precisa apenas ser complementada e validada na segunda atividade, antes da geração da especificação final.

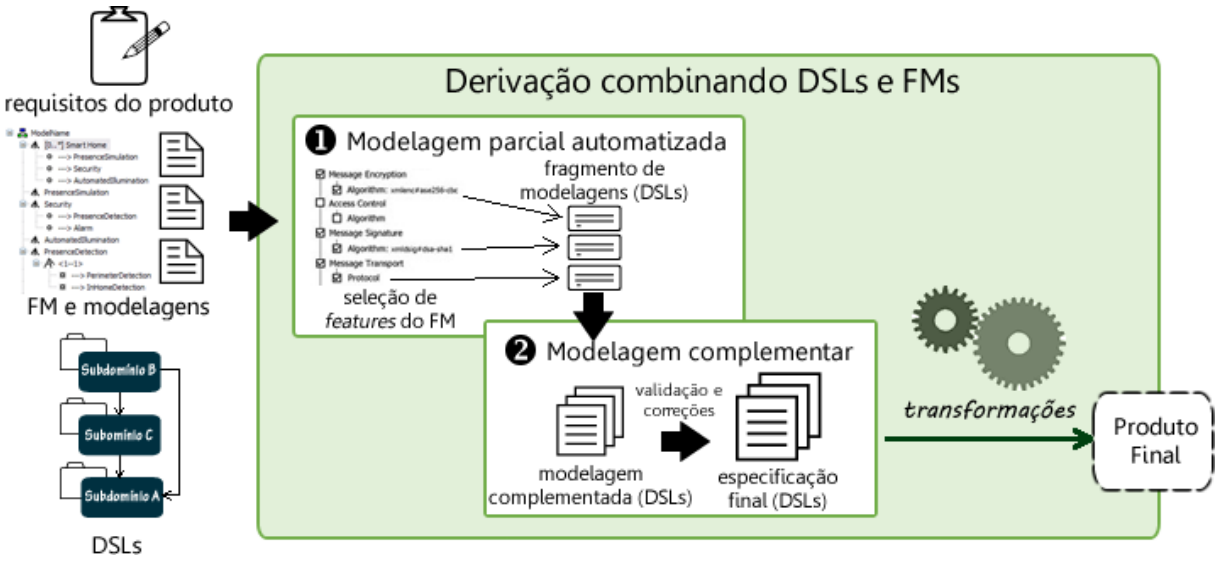


Figura 9. Fluxo de atividades na derivação combinando DSLs e FMs

Esta estratégia é mais indicada se deseja derivar produtos com característica bastante similar a outros já gerados na mesma abordagem, ou seja, quando já existem outras modelagens realizadas e queremos reaproveitar fragmentos dessas para gerar novos produtos. Assim, é necessário um trabalho a parte de reunir as modelagens existentes, identificar quais fragmentos poderão variar e construir um FM com *features* que enderecem tais fragmentos. Esse esforço inicial ocorrerá apenas na primeira utilização dessa derivação ou em caso de evoluções da LPS, nas demais, a derivação seguirá o fluxo padrão esquematizado na Figura 9.

Assim como na derivação apenas com DSLs, é necessário que seja fornecida uma entrada formada pelas DSLs e requisitos desejados para o produto. Contudo, além disso, um FM com seus respectivos fragmentos de modelagens também devem ser indicados como entrada nesta estratégia. A partir desses artefatos de entrada, a primeira atividade é gerar uma modelagem parcial do produto a partir da seleção de *features* do modelo de *features* (Figura 9 – Atividade 1). A segunda atividade (Figura 9 – Atividade 2) é reunir tais fragmentos, complementando e validando a modelagem, tal como realizado na estratégia utilizando apenas DSLs, para compor a especificação final, de forma a finalmente habilitar a transformação das DSLs para a geração de um produto final, similar à estratégia anterior.

3.3. Sumário

Este capítulo apresentou o método proposto nesta dissertação para lidar com o desenvolvimento de abordagens generativas com composição de DSLs. Foi apresentada uma visão geral da estrutura do método e detalhado os artefatos de entrada e saída de cada fase do método, assim como os papéis envolvidos. Foi apresentado também como integrar modelos de *features* e DSLs durante a engenharia de domínio e aplicação e apresentado diferentes estratégias de derivação de produtos em abordagens generativas com múltiplas DSLs.

4. ESTUDO DE AVALIAÇÃO DO MÉTODO DE DESENVOLVIMENTO DE ABORDAGENS GENERATIVAS COM COMPOSIÇÃO DE LINGUAGENS ESPECÍFICAS DE DOMÍNIO

Este capítulo apresenta o estudo exploratório realizado para avaliar o método proposto em uma abordagem generativa com composição de DSLs. A Seção 4.1 apresenta os objetivos do estudo e respectivas questões de pesquisas. A Seção 4.2 detalha o domínio investigado, ressaltando as razões de sua escolha. A Seção 4.3 apresenta as diferentes etapas do estudo e as seções seguintes apresentam os resultados do estudo: definição do escopo (Seção 4.4), aplicação do método (Seção 4.5), sumário dos artefatos produzidos (Seção 4.6) e discussão dos resultados (Seção 4.7). Por fim, a Seção 4.8 conclui destacando as ameaças à validade do estudo e suas principais contribuições.

4.1. Objetivos do Estudo e Questões de Pesquisa

O principal objetivo do estudo foi avaliar o método proposto no trabalho, através do desenvolvimento de uma abordagem generativa que envolvesse a composição de múltiplas DSLs. Em particular, o estudo exploratório foi desenvolvido com o objetivo de responder a segunda questão de pesquisa (QP2) desta dissertação (Capítulo 1), que questiona: "*Como implementar abordagens generativas que envolvam a composição de múltiplas linguagens específicas de domínio usando tecnologias atuais de engenharia dirigida por modelos?*". Com duas subquestões envolvidas: (QP2.1) "*Como a composição de linguagens específicas de domínio pode ser especificada e implementada durante a engenharia de domínio?*"; e (QP2.2) "*Como estratégias de derivação de produtos/sistemas de abordagens generativas que envolvam a composição de linguagens específicas de domínio podem ser implementadas na engenharia de aplicação?*".

Para respondê-la, o método desenvolvido e proposto neste trabalho para implementar abordagens generativas com composição de DSLs foi aplicado a um domínio real com o intuito de investigar a especificação e composição de DSLs durante a engenharia de domínio, e de descobrir formas de como utilizar estratégias derivação de produtos com composição de DSLs na engenharia de aplicação.

4.2. Seleção do Estudo-Alvo

De forma a escolher uma família de aplicações para o nosso estudo, foi exigido que ela devesse necessariamente atender aos seguintes pré-requisitos: (i) pertencer a um domínio real documentado e relatado na literatura; (ii) possuir um escopo com complexidade suficiente para ser seccionado em visões menores, possibilitando assim reuso de algumas partes; (iii) possuir requisitos, cuja representação, o modelo de *features* não fosse capaz de expressar; e (iv) ser de conhecimento dos pesquisadores (especialista do domínio) deste trabalho. Desse modo, foi escolhido o domínio de experimentos controlados em Engenharia de Software Experimental. Em especial, foi proposto aplicar o método proposto no desenvolvimento de uma abordagem generativa para produzir *workflows* de experimentos baseado na modelagem de DSLs. Tal abordagem foi proposta em (FREIRE, ACCIOLY, *et al.*, 2013) e desenvolvida em parceria com outros membros do nosso grupo de pesquisa. Assim, como parte deste trabalho, foi desenvolvida uma abordagem generativa para a modelagem de uma linha de experimentos controlados com múltiplas DSLs. Cada uma delas é destinada a modelar uma diferente perspectiva do domínio de experimentos controlados. As modelagens das DSLs foram utilizadas para derivar experimentos da LPS e, então, através de transformações entre modelos, gerar os *workflows* de procedimento de cada um dos seus participantes.

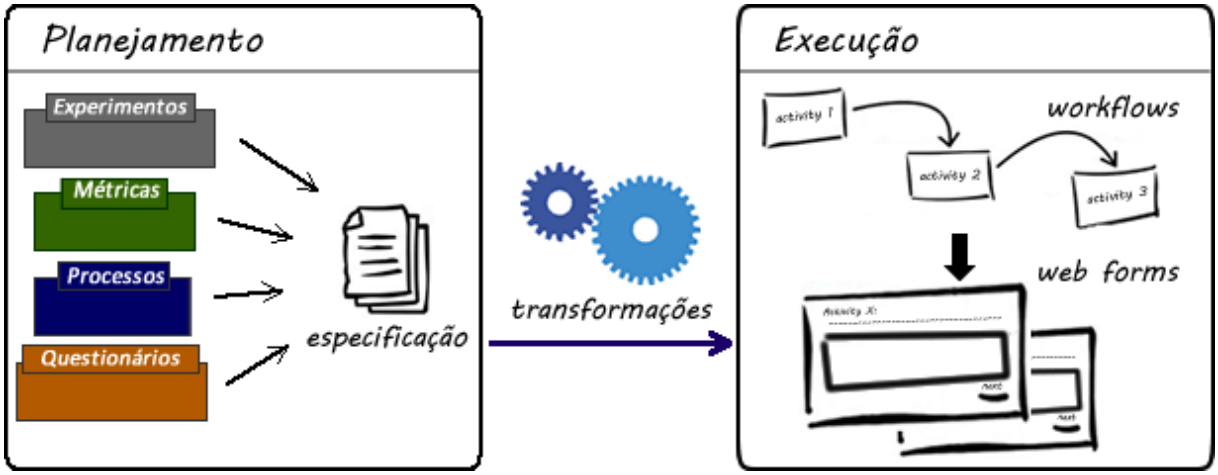


Figura 10. Visão geral da Abordagem de Experimentos proposta em *(FREIRE, ACCIOLY,* et al.*, 2013)*

A Figura 10 apresenta uma visão geral da abordagem dirigida por modelos para modelagem de experimentos controlados. Ela apresenta duas perspectivas:

planejamento e execução. A primeira agrega a modelagem do experimento a partir da especificação do experimento com seus respectivos processos, métricas e questionários. Após essa etapa, a modelagem resultante é usada para gerar os *workflows* e formulários web necessários para promover a execução do experimento e propiciar o monitoramento dos participantes durante a execução. O mapeamento entre as duas perspectivas é implementada por meio de transformações orientadas a modelo que não fazem parte do escopo deste trabalho. Originalmente, a abordagem de experimentos não contemplava a utilização de múltiplas DSLs e nem a definição formal das variabilidades da LPS. Nesse estudo, estamos interessados em aplicar o método proposto para desenvolver uma abordagem generativa com composição de DSLs que auxilia na modelagem de novos experimentos e permite a geração de *workflows* que os representam.

4.3. Etapas do Estudo

Para responder às questões levantadas, aplicamos o método proposto na dissertação em um cenário real com o objetivo de coletar resultados capazes de avaliá-lo. Para tal, inicialmente, foram identificados os requisitos que compõem o escopo a ser investigado para posteriormente aplicarmos a metodologia do estudo. Acrescentamos ainda uma etapa final para avaliação e discussão dos resultados. Assim, ficamos com a seguinte configuração do estudo:

(1) Definição de Escopo – inicialmente foram definidos os requisitos que contemplam o escopo de um experimento controlado para compreender o domínio investigado;

(2) Aplicação do Método – em seguida, foi aplicado o método no domínio escolhido seguindo suas fases e atividades. Durante a aplicação projetamos e implementamos uma abordagem generativa para a modelagem de experimentos controlados e geração de *workflows* de tais experimentos, usando as duas estratégias distintas identificadas;

(3) Avaliação e Discussões dos Resultados – por fim, foi realizada uma análise qualitativa do estudo com base nos resultados obtidos.

4.4. Definição do Escopo

Esta seção objetiva definir características do escopo explorado no estudo, no caso Engenharia de Software Experimental (ESE), mais especificamente, experimentos controlados. Um experimento controlado é uma técnica experimental que permite testar uma hipótese de pesquisa e os relacionamentos de causa e efeito entre diferentes variáveis (dependentes e independentes) envolvidas no ambiente de estudo. Um processo experimental é tipicamente composto por um conjunto de atividades e tarefas a serem conduzidas pelos participantes em cada tratamento. Cada tarefa consome e/ou produz artefatos relacionados com o estudo em questão. Experimentos controlados exigem um grande cuidado com planejamento para que possam fornecer resultados úteis e significativos (PFLEEGER, 1995). Para isso, é necessário definir um conjunto de métricas que estão associadas às tarefas, artefatos ou atividades de um processo e, em alguns casos, investigar o conhecimento prévio/acumulado dos participantes por meio da aplicação de questionários.

Baseados no *know-how* do nosso grupo de pesquisa na realização de experimentos controlados (CAMPOS NETO, BEZERRA, *et al.*, 2012; CAMPOS NETO, FREIRE, *et al.*, 2013), foi especificado um conjunto de requisitos iniciais que permitem modelar o projeto de um experimento controlado. Estes requisitos compõem os requisitos da abordagem generativa nessa etapa para atender a necessidade da área de propor um ambiente de suporte às fases de planejamento e execução de experimentos. Os requisitos, após elicitados, foram documentados e estão apresentados no Apêndice A deste documento. A definição do escopo a partir da documentação dos requisitos é importante para se entender o domínio investigado, além de fornecer os conhecimentos mínimos necessários para projeção e implementação da abordagem generativa, com todas as suas características comuns e variáveis, a serem desenvolvidas nas etapas seguintes.

4.5. Aplicação do Método

Um dos objetivos do método é identificar diferentes domínios no escopo durante a fase de projeto e, consequentemente, referências que serão implementadas na fase seguinte, por meio de composição. O método propõe ainda

explorar o uso de múltiplas DSLs para modelar o domínio de uma abordagem generativa, investigando também assim as limitações na expressividade dos modelos de *features*, assim como investiga alternativas para derivação de produtos usando DSLs. Nas subseções a seguir, apresentamos os resultados da aplicação do método neste estudo para cada uma das etapas do mesmo.

4.5.1. Visão geral da aplicação do método

A partir do nosso conhecimento sobre o domínio da abordagem generativa de experimentos controlados em engenharia de software (FREIRE, ACCIOLY, *et al.*, 2013), extraímos os requisitos do domínio (Apêndice A) e representamos as variabilidades em um modelo de *features* para ser fornecido como entrada do método, juntamente com os modelos arquiteturais do sistema web responsável pela instanciação e execução dos *workflows* de cada experimento. A partir desses artefatos, o método foi aplicado seguindo a sequência de atividades prevista em sua própria estrutura e produzindo novos artefatos a cada fase. A Figura 11 apresenta uma visão geral dos principais artefatos produzidos após a engenharia de domínio. Partindo dos modelos de entrada, o domínio de experimentos foi seccionado em subdomínios com relacionamentos entre si que foram documentados. Essa relação de subdomínios e sobreposições entre eles levaram a especificação de um conjunto de DSLs (uma para cada subdomínio) que, em conjunto com a especificação das transformações M2M, formam a abordagem generativa de experimentos controlados desenvolvida no estudo exploratório como resultado da engenharia de aplicação.

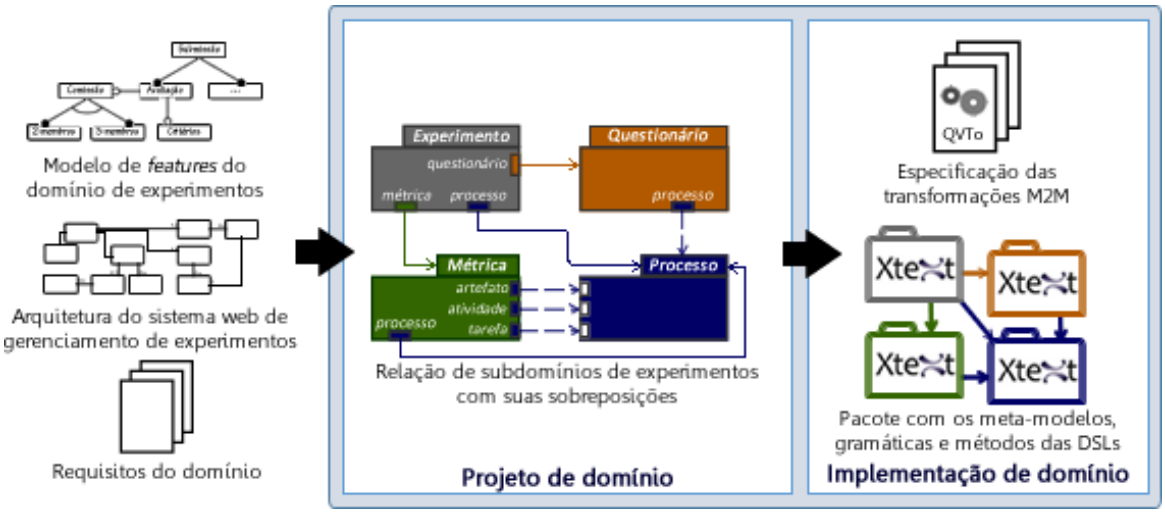


Figura 11. Visão geral da abordagem generativa de experimentos produzida

Após a execução da engenharia de domínio, obtemos como resultado uma abordagem generativa completa formada por um conjunto de quatro DSLs, construídas e especialmente preparadas para modelar experimentos contemplados pelo escopo da abordagem generativa e derivá-los a partir da execução das transformações também já especificadas. Irá variar na EA apenas o modo como esses experimentos são modelados, se usando apenas DSLs ou se combinando as DSLs com um FM construído especificamente para este fim. Para avaliar as duas possibilidades, realizamos a derivação de um mesmo produto duas vezes, aplicando em cada momento uma das estratégias e comparando qualitativamente os resultados produzidos por cada. Para isso, foi fornecido como entrada para a engenharia de aplicação os resultados alcançados na fase anterior (DSLs e especificação das transformações) e uma especificação com os requisitos do produto (no caso, experimentos) que queríamos derivar. Os detalhes sobre a aplicação do método e como da produção dos artefatos resultantes estão dispostos nas seções seguintes.

4.5.2. Projeto de Domínio

Na fase de projeto, analisamos a documentação do domínio que expressava as abstrações do domínio da abordagem generativa. Um modelo de *features* então foi construído para representar as variabilidades da abordagem de experimentos baseados e auxiliar na análise. Como resultado, identificamos quatro domínios principais. A Figura 12 apresenta o FM resultante do domínio de experimentos controlados destacando as variabilidades pertencentes a cada um dos domínios e a seguir comentamos cada um deles.

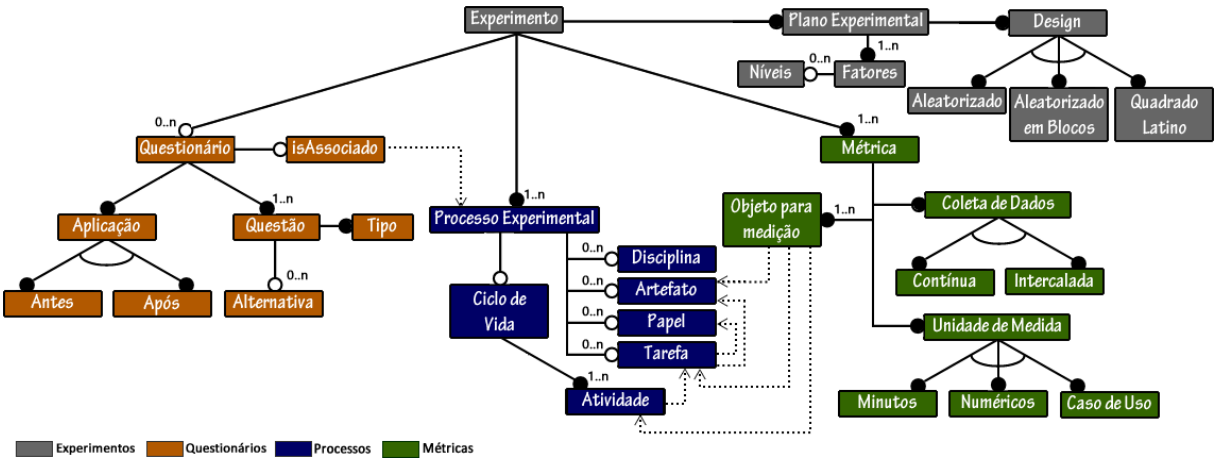


Figura 12. Modelo de *features* da LPS de Experimentos Controlados com destaque aos domínios identificados

Cada um dos domínios identificados representa um aspecto, uma visão, específica da LPS, ou família de sistemas, produzida, sendo possível de ser reutilizada. São eles:

(i) *Processo experimental* ─ que contém conceitos relacionados às atividades que são desempenhadas em cada tratamento de um experimento. Um processo experimental assemelha-se a um processo de desenvolvimento de software tradicional, sendo formando pelos mesmos conceitos bases, tais como disciplinas, artefatos, tarefas e papéis. Além disso, há a definição de um ciclo de vida que determina a ordem de execução das atividades;

(ii) *Métrica* ─ compreende as métricas que se deseja observar em uma experimentação, variando quanto à unidade de medição, a forma de coletar dados, o processo na qual está relacionada ou o tipo de elemento do processo que deseja mensurar (uma tarefa, um artefato ou uma atividade);

(iii) *Questionário* ─ domínio responsável pela definição de pesquisas que são aplicadas junto com o experimento para coletar dados de *feedback* complementares à análise dos resultados; um questionário é formado por questões de diferentes terminologias e pode ser aplicado antes ou após, respectivamente, da execução de um experimento ou um processo específico do experimento;

(iv) *Experimento* ─ por fim, o domínio de experimentos representa os elementos que descrevem o projeto experimental em si, trazendo consigo conceitos de o plano de experimento e conectar esse domínio aos demais.

Após a separação dos domínios, passamos a identificar as sobreposições entre os mesmos e classificá-las quanto ao grau de complexidade das referências. Para isso, enquanto especialista e conhecedor do domínio, confrontamos as especificações dos requisitos com as restrições representadas no modelo de *features*. Essa análise resultou na identificação de oito pontos de referências entre os domínios, ilustrados na Figura 13, e comentados a seguir.

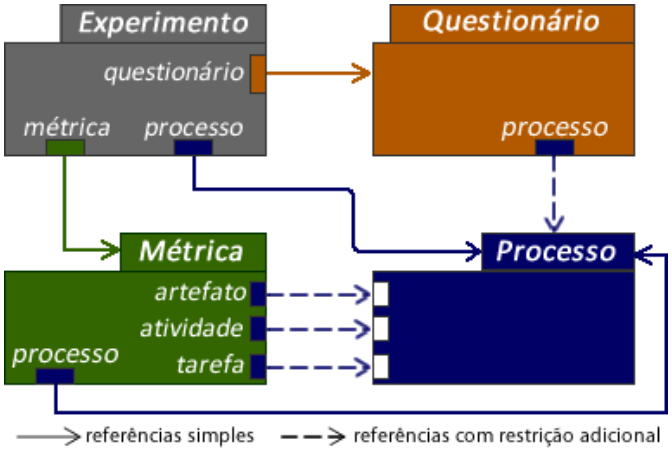


Figura 13. Esquema de sobreposições entre os domínios

O domínio de processos é o único a não fazer referência a nenhum outro. Já o domínio de experimentos precisa referenciar os processos de cada tratamento, as métricas que serão utilizadas para análise dos resultados e os questionários relacionados ao experimento. Por sua vez, o domínio de métricas está relacionado a um processo e precisa apontar quais artefatos, atividades ou tarefas desse serão considerados para a medição. Neste caso, identificamos um caso típico de referência com restrição adicional, onde o elemento referenciado na métrica precisa obrigatoriamente atender a restrição de ser um artefato, atividade ou tarefa existente no processo relacionado pela métrica como um todo. De mesmo modo, um questionário referenciado em um experimento pode também referenciar um ou mais processos referenciado pelo domínio de experimentos.

A separação de domínios também se baseou na arquitetura do motor de *workflows* responsável por executar os *workflows* do experimento para cada participante. O motor de execução é um sistema web sendo desenvolvido em outro trabalho de mestrado, que atua desde a importação de uma definição do experimento, a qual é gerada pela abordagem generativa do estudo exploratório desta dissertação, até a análise dos resultados. A ferramenta foi idealizada em uma arquitetura web baseada no padrão de projeto Modelo-Visão-Controlador (*Model-View-Controller* – MVC) (GAMMA, HELM, *et al.*, 1994), cujos componentes de controle definem atuações diferenciadas do motor de *workflows* para cada experimento, a partir do *upload* de um arquivo de definição de cada um dos *workflows* dos participantes. O arquivo contém as informações necessárias para montar os processos e instanciar os fluxos de execução com as atividades e tarefas a serem executadas. Após o *deploy* do experimento, o pesquisador deve ainda

configurar os arquivos para cada artefato de entrada indicado no arquivo de definição do experimento e cadastrar os participantes do experimento, atribuindo a cada um deles as devidas instâncias do processo definido no arquivo de configuração importado. Esta última atividade pode ser feita manualmente ou importando um arquivo de configuração do design do experimento. Ambos os arquivos, definição do experimento e configuração do design, são gerados a partir de uma seleção de *features* pertencentes aos domínios identificados na LPS, seja por meio de um modelo de *features* ou composição de DSLs, e posteriores transformações de modelo, discutida adiante na Seção 4.5.3.3.

A partir da especificação das DSLs é possível gerar automaticamente workflows para cada participante do experimento, com o intuito de coletar dados que são mensurados conforme as métricas estabelecidas. O processo de execução acontece em uma tela configurada para acomodar o processo ao qual cada participante foi relacionado, cada um tem acesso apenas aos formulários de resposta das atividades e tarefas as quais foi relacionado. A tela de resposta de cada atividade acontece na sequência que o pesquisador definiu na configuração do experimento, sendo que cada tela deverá ser composta por formulário de resposta de atividade e formulários de resposta das tarefas da atividade. Cada um destes formulários pode conter a opção para artefatos de saída, desde que seja indicado na configuração do experimento. Durante a execução dos formulários, o sistema armazenará as métricas indicadas, tais como o tempo de execução de cada participante. Ao final do processo, o sistema pode apresentar um novo formulário com questões de *feedback*, caso tenha sido definido pelo pesquisador.

4.5.3. Implementação do Domínio

Durante a etapa de implementação, foi identificada a necessidade de criação de DSLs para lidar com questões de limitação de expressividade do FM. O mesmo não foi capaz de expressar: (i) *atributos*, tais como, na *feature* “fator” que necessitaria de um atributo “tipo” para configurar o tipo de fator do experimento; (ii) especificar condições de *recursividade* que limitasse, por exemplo, a quantidade de fatores existente em experimento conforme o tipo de Design Experimental selecionado; e nem (ii) *referências* com restrições adicionais, como necessário no

domínio de processos. Nesse caso, a elaboração de DSLs, além de ampliar a expressividade da abordagem, pode ser usada como arquivos de configurações para os experimentos, possibilitando uma visão mais interessante aos usuários. Assim, foi gerada uma DSL para cada domínio, as quais estão relacionadas. A especificação de um experimento através de DSLs pode então gerar os arquivos de definição do experimento e configuração do *design* a ser importado pelo motor de *workflow*. As DSLs foram desenvolvidas utilizando o framework *xText* (http://eclipse.org/xtext), baseadas no meta-modelo *Ecore* do *Eclipse Modeling Framework* (EMF) (STEINBERG, BUDINSKY, *et al.*, 2008). São elas: (i) *ProcessDsl*, (ii) *MetricDsl*, (iii) *QuestionnaireDsl*, e (iv) *ExperimentDsl*.

4.5.3.1. Projeto individual das DSLs

A primeira DSL a ser implementada, a *ProcessDsl* (Listagem 1), é utilizada para definir os tratamentos, define as tarefas (*tasks*) necessárias para a realização de um teste experimental, tais tarefas são representadas na DSL, de forma a possuírem artefatos de entrada e saída bem como um papel responsável. No elemento raiz, *Process*, o atributo *name* define o processo. Além disso, é possível definir disciplinas, papéis, artefatos e tarefas associados ao processo, nos atributos *disciplines*, *roles*, *artifacts* e *tasks*, respectivamente. Por fim, o atributo *lifecycleElement* define os elementos que formam o ciclo de vida do processo que podem ser uma ou mais *Activity*. Esse último deve ainda relacionar no seu atributo *tasks*, as tarefas do processo que serão executadas na atividade.

Listagem 1. Gramática da *ProcessDsl*

```
grammar br.ufrn.dimap.ProcessDsl with org.eclipse.xtext.common.Terminals
generate processDsl "http://www.ufrn.br/dimap/ProcessDsl"

Model: (process+=Process);

Process: 'process' name=STRING '{'
            (('disciplines {' (disciplines+=Discipline)* '}'|
            'roles {' ('role {'roles+=Role'}')* '}'|
            'artifacts {' ('artifact {'artifacts+=Artifact'}')* '}'|
            'tasks {' ('task {'tasks+=Task'}')* '}'))*
            'lifecycle {' (lifecycleElement+=Activity)*'}'
         '}';

Discipline: 'discipline' name=STRING;
```

```
Role: 'name' name=ID
      'description' description=STRING;

Artifact: 'name' name=ID
          'description' description=STRING
         ('domain' domain=[Discipline])?;

Task: 'name' name=ID
       'description' description=STRING
      ('discipline' discipline=[Discipline])?
       'roles {' (rolesElement+=RoleElement)* '}'
       'artifacts {' (artifactsElement+=ArtifactElement)* '}'
      ('steps {' (steps+=Step)* '}')?;

RoleElement: roles=[Role]
             primacy=('primary'|'additional');

ArtifactElement: artifacts=[Artifact]
                 pinType=('input'|'output');

Step: 'name' name=ID
     ('previous' previous=[Step])?;

Activity: 'activity {'
             'name' name=ID
             'description' description=STRING
            ('tasks {'(tasks+=[Task])* '}'|
            ('roles {' (rolesElement+=RoleElement)* '}'
             'artifacts {' (artifactsElement+=ArtifactElement)* '}'))
            ('next' nextActivity+=[Activity])?
          '}';
```

A *MetricDsl* (Listagem 2) permite a especificação de métricas que serão coletadas durante um experimento para atender algumas das variáveis dependentes no estudo. Um experimento pode ser composto de uma ou mais métricas. O elemento raiz da linguagem é o *Metrics*, o qual é usado para definir cada uma das métricas que irão interceptar determinadas atividades ou tarefas do tratamento durante o experimento. Cada elemento possui um atributo *id* que é utilizado para identificar a métrica, um atributo *name* que deve permitir a identificação do nome da métrica e um atributo *relatesTo* que deve identificar o processo que a métrica deve interceptar. As informações adicionais sobre a métrica devem ser definidas no atributo *description*. A propriedade *form* representa o tipo de evento quantificado pela métrica. O *ColectType* do tipo *continuous* significa que a tarefa deve ser quantificada de forma ininterrupta, como um intervalo que vai do início até o fim da atividade/tarefa sendo medida. O *ColectType* do tipo *intercalated* permite que a coleta da métrica seja feita em intervalos que serão somados no final para o cálculo do valor total da medida. A propriedade *unit* é usada para indicar a unidade de medida da métrica.

Listagem 2. Gramática da *MetricDsl*

```
grammar br.ufrn.dimap.MetricDsl with org.eclipse.xtext.common.Terminals
generate metricDsl "http://www.ufrn.br/dimap/MetricDsl"

Model: (metrics+=Metrics)*;

Metrics: 'Metric' name=STRING 'relates' relatesTo=STRING '{'
            ('description' description=STRING)
            ('form' form=ColectType)?
            ('unit' unit=MetricUnit)?
            details=(ActivityMetric|TaskMetric|ArtifactMetric)
         '}';

ActivityMetric:
      ('activityBegin' activityBegin=STRING)
      ('activityEnd' activityEnd=STRING)?;

TaskMetric:
     'tasks' tasks+=STRING*;

ArtifactMetric:
     'artifacts' name=STRING artifacts+=STRING*;

enum ColectType returns ColectType:
     continuous = 'continuous' | intercalated = 'intercalated';

enum MetricUnit returns MetricUnit:
     minutes = 'minutes' |
     uc = 'uc' |
     numbers = 'numbers';
```

Por fim, o atributo *details* permite especificar o elemento a ser medido pela métrica, que pode ser uma atividade, uma tarefa ou um artefato do processo relacionado. Para atividades, as informações complementares correspondem aos campos *activityBegin* e *activityEnd* que correspondem, respectivamente, ao nome da atividade onde será iniciada a interceptação pela métrica e ao nome da atividade de fim da interceptação. Se a métrica pretende interceptar apenas uma atividade, o campo *activityEnd* deve ficar em branco. E caso não exista um fluxo válido entre essas atividades, apenas as duas nomeadas serão consideradas. Para tarefas, a única informação adicional necessária é a lista contendo os nomes das tarefas (separados por vírgula) no campo *tasks*. Para artefatos, os nomes dos artefatos relacionados à métrica devem ser descritos no campo *artifact*.

A *QuestionnaireDsl* (Listagem 3), permite especificar questionários com o objetivo de coletar *feedback* dos participantes do experimento. Esse retorno é importante para melhorar a qualidade dos dados obtidos durante o experimento.

Listagem 3. Gramática da *QuestionnaireDsl*

```
grammar br.ufrn.dimap.ExperimentDsl with org.eclipse.xtext.common.Terminals
generate questionnaireDsl "http://www.ufrn.br/dimap/QuestionnaireDsl"

Model: (questionnaire+=Questionnaire)*;

Questionnaire:
      'Questionnaire' name=STRING
     ('relates' relatesTo=STRING)?
      'type' questionnaireType=QuestionnaireType
      'Questions' '{'
         question+=Question*
      '}';

Question:
      name=STRING '{'
        ('description' description=STRING)
        ('type' type=AnswerType)
        ('required' req=INT)?
         'Alternatives' '{'
             alternatives+=STRING*
         '}'
     '}';

enum AnswerType returns AnswerType:
      Text = 'Text' |
      ParagraphText = 'Paragraph Text' |
      MultipleChoice = 'Multiple Choice' |
      CheckBoxes = 'Checkbox' |
      ComboBox = 'ComboBox' |
      Scale = 'Scale' |
      Grid = 'Grid' ;

enum QuestionnaireType returns QuestionnaireType:
      pre = 'Pre' |
      pos = 'Pos';
```

Cada questionário tem um atributo *name* e está relacionado com um ou mais processos (que representa os tratamentos experimentais) pelo atributo *relateTo*. Se o questionário está relacionado apenas com o experimento em si, não é necessário especificar este campo. O atributo *questionnaireType* é usado para definir se o retorno tem de ser recolhido antes ou depois da execução do experimento ou do processo. É uma enumeração cujos valores podem ser *Pre* ou *Pos*, respectivamente. Depois disso, as perguntas devem ser definidas usando o elemento *question*. Um questionário pode ter uma ou mais perguntas. Cada questão tem um atributo *name* para identificá-lo e um atributo *description* para explicar a questão. O atributo *type* é usado para selecionar o tipo de resposta. É uma enumeração (*AnswerType*) cujos valores podem ser: *Text*, *ParagraphText*, *MultipleChoice*, *ComboBox*, *CheckBox*, *Scala* ou *Grid*. Além disso, é preciso especificar se a questão é obrigatória de ser

respondida, através do atributo *req*, e especificar cada item de resposta nos atributos de alternativas, sempre que for necessário, de acordo com a *AnswerType* (tipo de resposta) selecionado.

Por fim, a *ExperimentDsl* (Listagem 4) foi definida para o contexto de ESE e permite, basicamente, definir o(s) tratamento(s) a ser(em) testado(s) pelo experimento e as variáveis que necessitam de controle durante a realização do mesmo. Um tratamento representa um método ou ferramenta que se deseja avaliar e pode ser composto por um ou mais fatores combinados. As variáveis de controle representam fatores que podem influenciar na aplicação de um tratamento e, desta forma, indiretamente o resultado do experimento. Os fatores devem ser definidos através do elemento *Factor* na linguagem, e cada fator pode possuir diferentes níveis de controle, que são definidos na linguagem através do elemento *Level*. Os fatores desejáveis devem ser sinalizados com o valor *True* na opção *isDesiredVariation* da linguagem.

Listagem 4. Gramática da *ExperimentDsl*

```
grammar br.ufrn.dimap.ExperimentDsl with org.eclipse.xtext.common.Terminals
generate experimentDsl "http://www.ufrn.br/dimap/ExperimentDsl"

Model: (elements+=ExperimentElement)*;

ExperimentElement:
       'Experiment' name=STRING
      ('Process' process +=STRING*)?
      ('Metric' metrics +=STRING*)?
      ('Questionnaire' questionnaires+=STRING*)?
      ('Experimental Plan' experimentalPlans+=ExperimentalPlan*)?;

ExperimentalPlan:
       'Design' name=STRING
       'type' type=DesignType
       '{'
          (factor+=Factor)*
       '}';

Factor: 'Factor' name=STRING
         'isDesiredVariation' isDesiredVariation = ('True'|'False')
         (level+=Levels)* ';';

Levels: 'Level' name=STRING ';';

enum DesignType returns DesignType:
      CDR = 'CRD - Completely Randomized Design' |
      RCDB = 'RCBD - Randomized Complete Block Design' |
      LS = 'LS - Latin Square';
```

Além da definição relatada anteriormente, o plano experimental deve dizer como os testes experimentais devem ser organizados e executados, ou seja, qual será o design estatístico do experimento (PFLEEGER, 1995). A linguagem apresenta o elemento *type* que permite a seleção de um dentre três tipos de *design* de experimentos disponíveis na linguagem que são o design completamente aleatorizado (*CRD – Completely Randomized Design*), design completamente aleatorizado em blocos (*RCBD – Randomized Complete Block Design*) e o quadrado latino (*LS – Latin Square*). Foram incluídos apenas os principais tipos de designs identificados na fase de análise. Existem diversos outros tipos de *designs* estatísticos de experimentos na literatura que não são ainda suportados pela abordagem de experimentos estudada e consequente não foram incluídos na linguagem.

4.5.3.2. Composição de DSLs

Após apresentar as gramáticas das DSLs, passamos a descrever a codificação dos pontos de sobreposições identificados na fase anterior. A codificação dessas referências é importante para que a abordagem generativa de modelagem de experimentos defina as regras de restrições dos modelos e validá-los. As modelagens são utilizadas para derivar novos experimentos.

A codificação utilizou recursos do próprio *xText* e alguns recursos adicionais de validações escritos em linguagem Java. Nativamente, o *xText* oferece a possibilidade que um meta-modelo importe um outro como forma de implementação de referência explícita entre as DSLs. Como estamos trabalhando com DSLs baseadas no meta-modelo *Ecore*, o *xText* possui um gerador de modelos para cada gramática criada, responsável pela geração de um meta-modelo *Ecore* equivalente. Para realizar a importação, o que precisa ser feito é informar, na gramática que se deseja criar a referência, o caminho para o modelo *Ecore* que se pretende importar e modificar a implementação do gerador de modelos da linguagem para incluir o *Ecore* importado no seu meta-modelo. A Listagem 5 ilustra a sintaxe de importação (abreviando o caminho do meta-modelo a ser importando) usada para referenciar as DSLs de processos, métricas e questionários na DSL de experimentos.

Listagem 5. Fragmento da gramática da *ExperimentDsl* realizando a importação dos metamodelos *Ecore* das DSLs de *ProcessDsl* e *MetricDsl*

```
grammar br.ufrn.dimap.ExperimentDsl with org.eclipse.xtext.common.Terminals

import "platform:/resource/…/…/ProcessDsl.ecore" as processDsl
import "platform:/resource/…/…/MetricDsl.ecore" as metricDsl
import "platform:/resource/…/…/QuestionnaireDsl.ecore" as questionnaireDsl

generate experimentDsl "http://www.ufrn.br/dimap/ExperimentDsl"
```

Com a importação, cada meta-modelo referenciado passa a ser reconhecido por um *alias*, tornando assim possível referenciar de forma explícita os elementos do outro modelo. A Listagem 6 mostra que agora a *ExperimentDsl* passa a armazenar valores de referências nos atributos *process*, *metrics* e *questionnaires*, respectivamente representados pelos elementos *processDsl::Process*, *metricDsl::Metrics* e *questionnaireDsl::Questionnaire*.

Listagem 6. Fragmento da gramática da *ExperimentDsl* apontando outras DSLs

```
ExperimentElement:
       'Experiment' name=STRING
      ('Process' process +=[processDsl::Process]*)?
      ('Metric' metrics +=[metricDsl::Metrics]*)?
      ('Questionnaire' +=[questionnaireDsl::Questionnaire]*)?
```

Para a *MetricDsl* e *QuestionnarieDsl* partimos de uma lógica análoga à utilizada em *ExperimentDsl*, porém, como visto, nestas linguagens há algumas referências com restrições adicionais para as quais precisam ser criadas rotinas extras de validação. É o caso da referência a artefatos, tarefas ou atividades que dependem do processo ao qual a métrica está relacionada. Em gramáticas baseadas em *Ecore*, o gerador de modelos cria também classes em Java equivalente a cada elemento da DSL, além de classes auxiliares com recursos específicos de formatação, validação, *etc*. Usamos os recursos de validadores disponíveis pelo *xText* para codificar restrições adicionais nesse caso. As regras são implementadas usando a linguagem Java e um exemplo desta é apresentado na Listagem 7. O objetivo é recuperar todas as *tasks* do processo relacionado e iterar sobre as mesmas de modo que seja possível fazer as validações das restrições adicionais.

Listagem 7. Método para validação do elemento *TaskMetric* de *MetricDsl*

```
public class MetricDslJavaValidator extends AbstractMetricDslJavaValidator
{

  @Check
  public void checkTaskIsValid(Metrics metrics) {
    boolean isValid = false;
    if (metrics.getDetails() instanceof TaskMetric) {
      for (Task _task : metrics.getRelatesTo().getTasks())
        for (Task task : ((TaskMetric)metrics.getDetails()).getTasks())
          if (task.equals(_task.getName())) isValid = true;
    }
    if (!isValid) error("Invalid task reference.",
                        MetricDslPackage.Literals.METRICS__DETAILS);
  }

}
```

A classe *MetricDslJavaValidator* foi gerada pelo *xText* após a compilação da gramática e nela criamos os métodos de validação. A Listagem 7 demonstra um fragmento dessa classe com foco no método *checkTaskIsValid(Metrics metrics)*, criado para validar tarefas na DSL de métricas. De modo análogo, outros dois métodos foram construídos para validar atividades e artefatos.

4.5.3.3. Geração Automatizada dos Workflows no Motor de Execução

Após a definição do experimento utilizando as DSLs, a especificação deve ser transformada para geração dos arquivos de configuração e de especificação dos *workflows*. Esses arquivos são usados como entrada para o sistema web, a partir de uma operação de *upload*, para que o mesmo possa gerar, executar e monitorar os *workflows* dos participantes do experimento, com base em informações provenientes das especificações realizadas com as DSLs. A especificação do experimento através do uso de DSLs possibilita a geração de *workflows* executáveis que representam os tratamentos que devem ser distribuídos para cada participante do experimento, de acordo com as variabilidades do experimento escolhidas. Estes *workflows* são automaticamente gerados, através de uma abordagem dirigida por modelos, que envolve também a especificação de transformações dirigidas por modelos e gera como saída *workflows* customizados para cada participante do experimento. Os *workflows* gerados são responsáveis por guiar cada participante através das

atividades e tarefas necessárias para conduzir o experimento e extrair os dados relacionados às métricas especificadas.

A Figura 14 apresenta o fluxo de geração automatizada de *workflows* a partir dos resultados obtidos pela derivação de um experimento na abordagem generativa utilizando as DSLs. A especificação final do experimento derivado da abordagem é transformada para geração dos (i) arquivos de configuração, responsáveis por determinar a organização da execução do experimento de acordo com o design estatístico definido, fazendo assim a distribuição dos tratamentos e a aleatorização dos participantes; e (ii) especificações dos *workflows*, que são responsáveis por gerar os *workflows* relativos à coleta de dados dos participantes, tal geração ocorre a partir das informações dos processos, questionários e métricas definidos.

Figura 14. Geração dos *workflows* a partir da especificação das DSLs

Para iniciar a execução do experimento, o pesquisador necessita fazer o *upload* dos arquivos gerados na etapa anterior (especificações dos *workflows* e arquivos de configuração), que representam o experimento completo, no ambiente web do motor de execução. Na sequência, o pesquisador deve executar algumas configurações simples, tais como, registrar participantes e carregar os artefatos de entrada especificados para as atividades dos processos envolvidos. Por fim, ele pode habilitar a execução do experimento. Deste ponto em diante, cada participante do experimento pode dar início à execução das suas atividades seguindo o fluxo descrito no seu *workflow*. Durante a execução, as métricas são coletadas de acordo com a especificação e os dados coletados ficam disponíveis para a fase de análise. A execução não foi explorada neste estudo, pois está fora do escopo deste trabalho. Na seção a seguir, são expostos mais detalhes das transformações implementadas.

4.5.3.4. Implementação das Transformações de Modelos

A gramática das DSLs abstrai as variabilidades existentes em cada domínio, as quais depois de modeladas necessitam passar por transformações para poder gerar os arquivos de configuração que irão ser lidos pelo motor de execução para instanciação dos *workflows*. Estes *workflows* são automaticamente gerados através de uma abordagem dirigida por modelos. O processo de transformação envolveu a realização de transformações M2M (modelo para modelo) e transformações M2T (modelo para texto). Primeiramente, os arquivos textos com as modelagens do experimento, das métricas, dos processos e dos questionários (gerados, respectivamente, utilizando cada uma das DSLs) são convertidos em arquivos *Ecore* que representem modelos (gerado automaticamente pelo *xText*) em conformidade com os meta-modelos *Ecore* de cada uma das DSLs (os meta-modelos podem ser visto no Apêndice B). Estes arquivos *Ecore* passam então por duas transformações: (i) a transformação M2M é responsável por gerar as especificações dos *workflows* a partir das informações dos processos, questionários e métricas definidos; e (ii) a transformação M2T que é responsável por gerar os arquivos de configuração a partir de informações do experimento.

A linguagem QVTo (*Operational* QVT) foi usada para implementar as transformações M2M e para as transformações M2T foi usada a linguagem de *templates Acceleo* e a linguagem de programação Java. O meta-modelo de *workflow* utilizado é baseado no esquema jBPM *Process Definition Language* (JPDL), com mais elementos que o original, os quais são específicos para o domínio de processos da engenharia de software. A Tabela 1 apresenta o mapeamento entre os elementos nas três etapas da abordagem: elementos especificados nas DSLs, no modelo de *workflow* e no motor de execução. O elemento *Process* da *ProcessDsl* é mapeado como um elemento *process-definition* no modelo de *workflow*, finalizando como uma instância do workflow no motor de execução. Cada elemento *Activity* na linguagem de processos é mapeado como um elemento *task-node* no modelo de workflow, que é mapeado para um formulário web responsável por apresentar as atividades e suas tarefas ao participante durante o experimento. Cada atributo *next* do elemento *Activity* é mapeado para um elemento *transition* no modelo de *workflow* e um botão de transição no formulário web do motor de execução. O elemento *Metric* de *MetricDsl* é mapeado para um par de elementos no sistema de *workflow*: um elemento *Event* e

um *Action*. Eles são mapeados para ações implementadas como métodos Java no motor de execução. Os elementos *Artifact in* e *Artifact out* são mapeados em elementos similares no modelo de *workflow*. O elemento *Artifact in* representa um link para *download* no formulário da atividade correspondente, e o elemento *Artifact out* representa um link para *upload* no formulário web da respectiva.

Tabela 1: Mapeamento entre elementos

DSL	***Workflow Model***	***Web Application – Engine***
Process	*process-definition*	*workflow instance**
Activity	*task-node*	*web-form*
Task	*task*	*------*
next (Activity)	*transition*	*transition button*
Metric (activity, task, artifact)*	*(Event + action)*	*(java method)*
Artifact in	*artifact (type in)*	*link to download*
Artifact out	*artifact (type out)*	*upload form*
artifact inout	*artifact (type inout)*	*link to download + upload form*
participant	*role*	*users**

4.5.4. Derivação de Produtos

Nosso estudo exploratório investigou duas diferentes estratégias de derivação de produto prevista pelo método. A primeira usando apenas DSLs (Seção 4.5.4.2) e a segunda usando uma abordagem combinada de DSLs e FMs (Seção 4.5.4.3). Antes de apresentamos os resultados de cada uma das estratégias, apresentamos o experimento base utilizado para avaliar as estratégias (na Seção 4.5.4.1). Tal experimento envolveu a comparação de duas linguagens de desenvolvimento e foi adaptado de um exemplo didático do (WOHLIN, RUNESON, *et al.*, 2000).

4.5.4.1. Especificação de um Experimento Controlado

O objetivo do experimento (WOHLIN, RUNESON, *et al.*, 2000) é realizar uma comparação da produtividade no desenvolvimento usando-se as linguagens de programação Java e C++. O experimento considera, além das próprias linguagens, distintos sistemas e participantes como fatores de controle. Para isso, foram utilizados dois sistemas de níveis de complexidade diferentes. O primeiro, uma Agenda de Contatos (*Phonebook*), com reduzido número de casos de uso, e o segundo, um Sistema de Gerenciamento de Eventos (*EventManagement*), de maior complexidade. Os *participantes* do experimento foram aleatorizados, assim como os

sistemas e as linguagens, de modo que cada um pudesse executar uma mesma sequência de atividades com os dois sistemas, nos dois tratamentos (Java e C++), constituindo assim um design experimental aleatorizado em blocos.

Para realizar a comparação, os participantes devem seguir um processo experimental em cada tratamento no qual recebem um artefato de entrada já contendo a especificação dos casos de uso do respectivo sistema sorteado, e a partir daí tem que executar as seguintes atividades, com a linguagem, também sorteada:

- *Projetar casos de uso*: envolve as tarefas de projetar a arquitetura do sistema, tomando decisões de projeto para as tarefas de desenvolvimento e guiadas pelo modelo de casos de uso recebido como entrada, envolve também o projeto dos protótipos de tela do sistema;
- *Implementar casos de uso*: envolve a tarefa de implementar os artefatos (classes, interfaces) definidos na atividade anterior;
- *Testar casos de usos*: envolve realizar as tarefas de elaborar e executar os casos de testes para cada caso de uso implementado.

O conjunto dessas atividades, tarefas e artefatos constitui o processo simples utilizado para execução desse experimento, tendo como único papel desse processo os próprios participantes. O processo é parte do experimento, assim como as métricas, que também se relacionam ao processo e ou parte deste. Nesse experimento, foram consideradas três métricas para análise: (i) o tempo gasto para projetar os casos de usos; (ii) o tempo gasto para codificar cada caso de uso; e (iii) o tempo gasto para elaborar e executar os casos de testes.

Além disso, o experimento prevê a aplicação de um questionário de experiência após a execução do processo experimental, com perguntas sobre o nível de experiência dos participantes.

4.5.4.2. Derivação usando apenas DSLs

Esta seção apresenta os resultados para a estratégia de derivação utilizando apenas as DSLs projetadas. O resultado dessa derivação resultou na modelagem final do experimento controlado apresentado na seção anterior, que utilizou as quatros DSLs projetadas no estudo. Outros experimentos foram modelados em

trabalhos anteriores (CAMPOS NETO, FREIRE, *et al.*, 2013; FREIRE, ACCIOLY, *et al.*, 2013). A seguir são apresentados os resultados desta estratégia.

A partir da definição do escopo do experimento que queremos derivar, utilizamos as quatro DSLs implementadas na Engenharia de Domínio para modelar as *features* do experimento. A Figura 15 apresenta o resultado da modelagem realizada no ambiente *xText* para o experimento. A modelagem foi realizada usando a DSL de experimentos, mas nela é possível observar referências a processos, métricas e questionários modelados em outras DSLs e que precisam ser validadas.

```
Experiment "Comparison of Java and C++"
   Process SystemProcess
   Metric  ProjectTime CodeTime TestTime
   Questionnaire Experience
   Experimental Plan Design "ComparisonPlanning"
   type RCBD – Randomized Complete Block Design {
        Factor "ProgammingLanguage" isDesiredVariation True
            Level "Java";
            Level "C++";
        ;
        Factor "Subject" isDesiredVariation False;
        Factor "System" isDesiredVariation False
            Level "Phonebook";
            Level "EventManagement";
        ;
    }
```

Figura 15. Modelagem do experimento de comparação de linguagens de programação no ambiente *xText*

A composição utilizou recursos da própria ferramenta, nesse caso o ambiente *xText*, que apresentam recursos de navegação, manutenção, alertas e guias. A navegação é um recurso de visualização e comunicação entre os modelos. No *xText* esse recurso não chega a ser tão sofisticado, porém é suficiente para estabelecer um *link* entre uma linguagem e outra a partir dos pontos de referências. Por ser um *plug-in* da plataforma Eclipse, o *xText* utiliza os mesmos recursos de navegação de código da IDE, o que não chega a fornecer visões gráficas apropriadas, porém apresenta recursos de interação.

A verificação de manutenção é o ponto chave da aplicação do método, pois a partir dessa verificação é possível verificar se as restrições às consistências estão sendo mantidas ou violadas. Essa checagem de consistência é consequência das codificações realizadas durante a especificação e se torna visível a partir dos recursos nativos de alertas do *xText*. O *xText* pode apresentar tanto *warning* como

error, além de *pop-ups* com sugestões de valores durante a digitação (Figura 16) ou sugestões de consertos (Figura 17). Esse modo de apresentação já entra no âmbito das guias, por meio da apresentação de *pop-ups*.

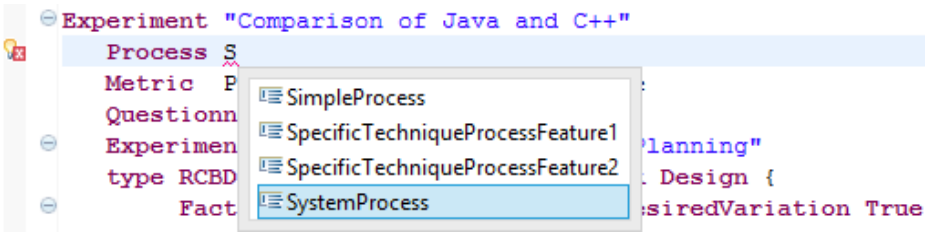


Figura 16. Demonstração de um *pop-up* com sugestão de referências

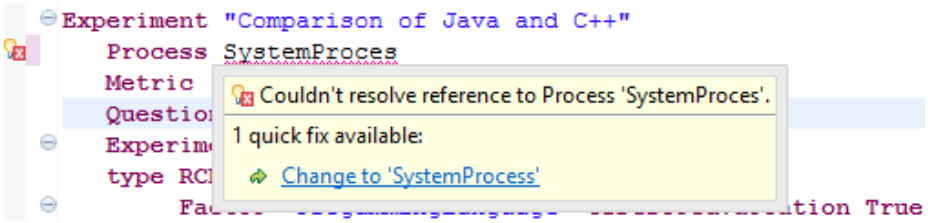


Figura 17. Demonstração de um *pop-up* com *error* e sugestão de conserto

A Figura 18 apresenta um fragmento da modelagem do processo, com destaque ao seu ciclo de vida, que define a sequência de execução das atividades. Ao todo foram definidas três atividades, cada um referenciando uma ou mais tarefas previamente definidas. O objetivo do processo é indicar um conjunto de tarefas de projeto, implementação e teste a serem executadas pelos participantes para comparar as linguagens avaliadas pelo experimento.

```
lifecycle {
    activity {
        name UseCaseProject description "Project of Use Case"
        tasks { DesignClassDiagram DesignScreens }
        next UseCaseImplementation
    }
    activity {
        name UseCaseImplementation description "Implementation of Use Case"
        tasks { ImplementUseCase }
        next UseCaseTest
    }
    activity {
        name UseCaseTest description "Perform Test of Use Case"
        tasks { CreateTestCases RunTests }
    }
}
```

Figura 18. Modelagem do ciclo de vida do processo experimental no ambiente *xText*

Os artefatos de entrada e saída estão indicados na definição das tarefas. A Figura 19 mostra exemplo da definição de uma tarefa.

```
task {
    name DesignClassDiagram description "Design Class Diagram"
    roles { Subject primary }
    artifacts { UseCaseSpecfication input
                ClassDiagram output }
}
```

Figura 19. Definição de uma tarefa usando a linguagem de processos no ambiente *xText*

A modelagem deve obedecer às restrições do domínio, uma delas diz respeito à questão dos artefatos bem-definidos. Para este caso, o *xText* utiliza a própria gramática da DSL para confrontar a sintaxe utilizada na modelagem a fim de verificar se há atributos em falta ou sintaxe incorreta. No caso de falha são apresentados alertas de *error,* como na Figura 20, quando na modelagem de uma *task* na *ProcessDsl* não foi informado o seu *name* antes do atributo *description*. Observe que a mensagem de *error* apresenta orientações para a correção. Em casos em que se requer a apresentação de mensagens customizadas, podem ser criadas ainda rotinas de validação, como as realizadas para restrições adicionais, apresentadas a seguir.

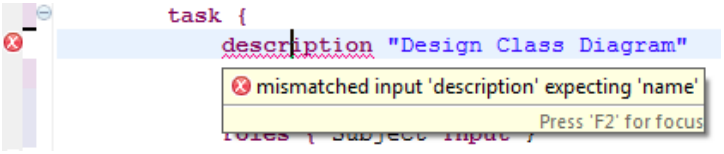


Figura 20. *Pop-up* indicando artefatos não bem-definido em *ProcessDsl*

As métricas foram modeladas usando a linguagem *MetricDsl* e os resultados são apresentados na Figura 21. As três métricas são de medição contínua e medidas em minutos. Além disso, elas referenciam as tarefas do processo experimental para medir o tempo de projeto, codificação e teste, respectivamente.

```
Metric "ProjectTime" relates SystemProcess {
    description "Project Time"
    form continuous unit minutes
    tasks "SystemProcess.DesignClassDiagram" "SystemProcess.DesignScreens"
}
Metric "CodeTime" relates SystemProcess {
    description "Time taken to answer a question correctly"
    form continuous unit minutes
    tasks "SystemProcess.ImplementUseCase"
}
Metric "TestTime" relates SystemProcess {
    description "Time taken to answer a question correctly"
    form continuous unit minutes
    tasks "SystemProcess.CreateTestCases" "SystemProcess.RunTests"
}
```

Figura 21. Modelagem das métricas do experimento utilizando a *MetricDsl* no ambiente *xText*

Conforme visto anteriormente, há a ocorrência de referências com restrições adicionais em *MetricDsl*, devido a cada métrica se restringir a referenciar apenas tarefas, atividades ou artefatos pertencentes ao processo referenciado pela métrica completa. Esse tipo de restrição, no passo atual, culmina em efeitos similares de apresentação de *pop-ups* aos demais tipos de restrições. O que muda apenas é a mensagem de exibição de erro que será personalizada, conforme implementação do método de validação realizada na especificação. A Figura 22 exemplifica um caso de violação a essa restrição durante a modelagem da métrica *CodeTime*. No exemplo, ela está relacionada ao processo *SimpleProcess*, mas referencia a *task DevelopTechnicalVision*, pertencente a um outro processo, o *OtherProcess*. Como resultado da execução da rotina de validação criada na fase de implementação, será apresentado um *pop-up* com uma mensagem de erro customizado. O erro é apresentado mesmo que a tarefa exista no processo *OtherProcess*, porém não existe no processo *SimpleProcess*, relacionado, não sendo assim uma tarefa válida para o cenário. A mesma mensagem de erro não é apresentada na segunda métrica, *TestTime*, pois diferente da primeira, as duas *tasks* referenciadas existem em *SimpleProcess*, sendo assim ambas válidas.

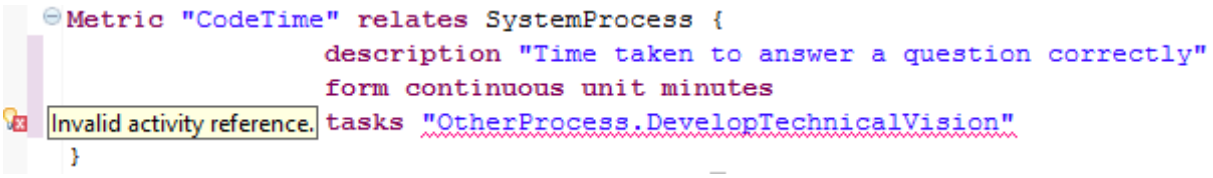


Figura 22. Modelagem de uma métrica com destaque à violação de restição adicional na modelagem

O último tipo de restrição é a de estilo. A especificação desse tipo também é feita usando-se métodos de validação, porém, não foram identificadas no nosso estudo exploratório. Apenas para fins de demonstração, poderíamos criar um estilo que impedisse que nomes de processos fossem iniciados com letras minúsculas. Como esse não seria um estilo obrigatório, a violação desse estilo implicaria apenas em um *warning*. A Figura 23 ilustra a apresentação dessa violação de restrição.

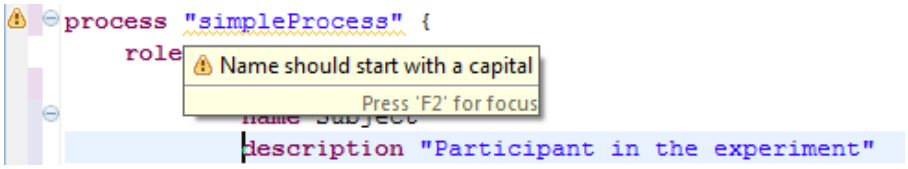


Figura 23. *Pop-up* indicando a violação de uma restrição de estilo

```
Questionnaire "Experience" relates SystemProcess type Pos
    Questions {
        "LevelExperience" {
            description "What is your experience using this technology?"
            type Multiple Choice
            required 1 Alternatives {
                "Start"
                "Professional"
                "Expert" }
        }
        "TimeExperience" {
            description "How long experience with this technology?"
            type Text
            required 1
        }
    }
```

Figura 24. Modelagem do Questionário de Experiência utilizando a *QuestionnaireDsl*

Por fim, a Figura 24 apresenta o questionário de *feedback* modelado com a *QuestionnaireDsl* para ser aplicado após a execução do processo *SystemProcess*. O mesmo possui duas questões, a de múltipla-escolha sobre o nível de experiência e outra subjetiva sobre o tempo de experiência dos participantes.

4.5.4.3. Derivação combinando DSLs com FM

Para avaliar a estratégia que combina FMs com DSLs, inicialmente identificamos algumas variabilidades comuns aos nossos experimentos controlados e as reunimos em grupos, tais como, os mostrado na Tabela 2, classificando-as quanto a sua multiplicidade, para poder representá-las em modelo de *features*.

Tabela 2. Algumas variabilidades encontradas nas modelagens dos experimentos controlados

Grupo	#	Variabilidade	Tipo
Tipo de Design	01.0	Completamente Aleatorizado	Alternativa
	01.1	Completamente Aleatorizado em Blocos	Alternativa
	01.2	Quadrado Latino	Alternativa
Métricas	02.0	Tempo de Resposta	Opcional
	03.0	Tempo de Projeto	Opcional
	04.0	Tempo de Codificação	Opcional
	05.0	Tempo de Teste	Opcional
Questionários	06.0	Experiência	Opcional
	07.0	Escolaridade	Opcional
Questões de Experiência	08.0	Nível de Experiência	Opcional
	09.0	Tempo de Experiência	Opcional
	10.0	Detalhes da Experiência	Opcional
Questões de Escolaridade	11.0	Nível de Escolaridade	Opcional
	12.0	Tema de Pesquisa	Opcional

No estudo, foi utilizado o framework *FeatureMapper* (http://featuremapper.org/) para representar as variabilidades identificadas através de um modelo de *features*. Esse framework permite definir um modelo de *features* agrupando-as em grupos e informando uma cardinalidade máxima e mínima, tal como, a multiplicidade identificada para cada variabilidade. A Figura 25 apresenta o modelo de *features* resultado dessa representação usando o *FeatureMapper*.

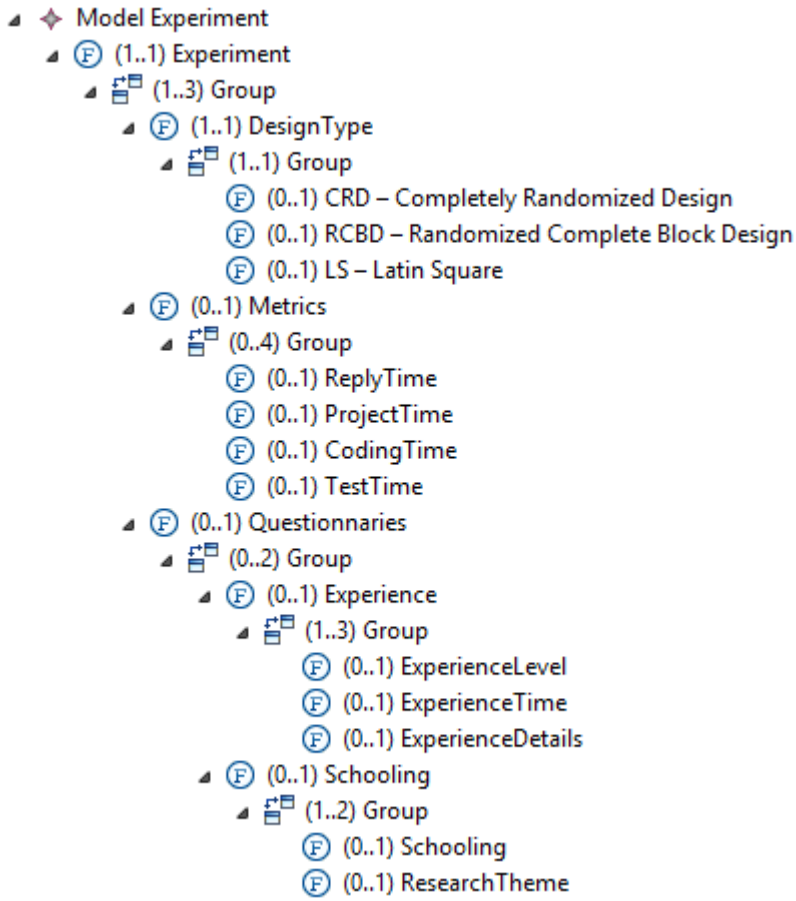


Figura 25. Modelo de Features construído no *FeatureMapper* para representar variabilidades dos experimentos

O passo seguinte é representar a modelagem dos artefatos da LPS usando arquivos XMI (Figura 26) dos meta-modelos para poder ser mapeado na ferramenta. Em seguida, usando o framework *FeatureMapper*, criamos um modelo de mapeamento para ligar as features do modelo de features com os artefatos modelados e representados em arquivos XMI. O *plug-in* possui um modo de visualização de mapeamento, chamado de *MappingView*, tal como mostrado na Figura 27, onde do lado esquerdo é carregado o modelo de *features* que se deseja endereçar as *features* e do outro lado o arquivo de modelo XMI representando os artefatos que são endereçados. O mapeamento é feito pela seleção de uma *features* (ou expressões/junções delas) de um lado e os respectivos artefatos que estarão no

modelo final, do outro lado, caso tais *features* sejam selecionadas. A Figura 27 mostra o exemplo da configuração das *features* "*Scholling*" e "*ResearchTheme*" que, respectivamente, quando selecionadas, implicam na inserção de um "*Questionnaire Scholling*" (questionário de escolaridade) que possua uma "*Question ResearchTheme*" (questão sobre o tema de pesquisa do participante).

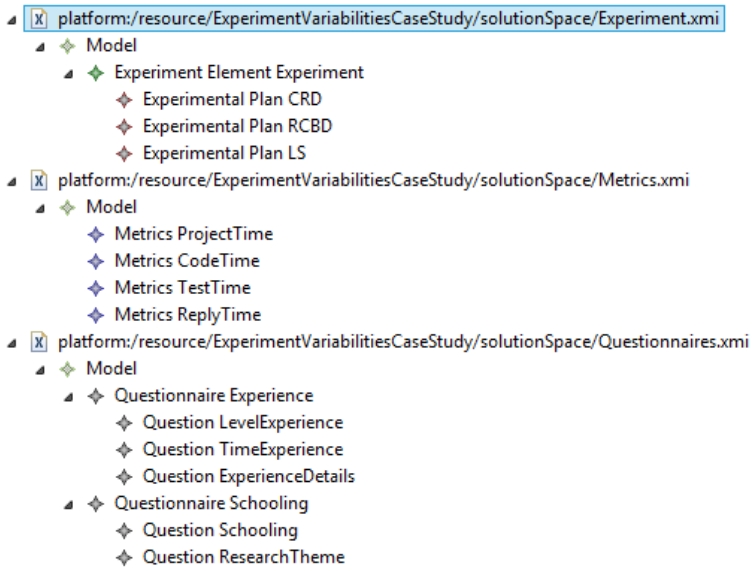


Figura 26. Representação da modelagem dos artefatos da LPS usando arquivos XMI

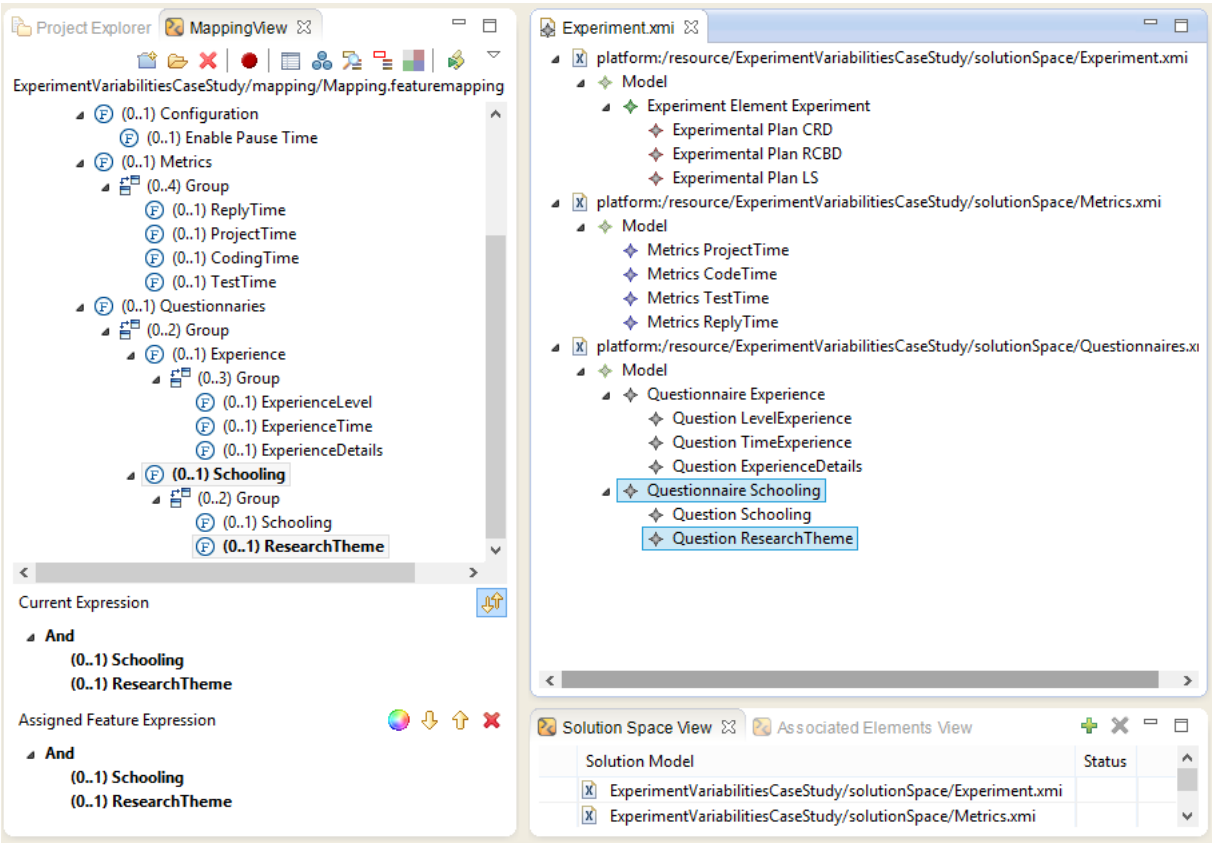


Figura 27. Visão de mapeamento do *FeatureMapper*

Após a definição do mapeamento, deve ser gerado um modelo de variação usando o *FeatureMapper*, chamado de *variant* pelo plug-in, para selecionar quais *features* queremos incluir no nosso experimento a ser gerado. A modelagem resultante é a junção dos fragmentos de modelagem selecionados e que ainda necessita de ajustes antes de ser transformado para os workflows. Para avaliar a derivação, selecionamos um conjunto de *features* para derivar um experimento com as mesmas especificações do derivado com a estratégia anterior. A Figura 28.a mostra o resultado do processamento da derivação para este conjunto de *features*.

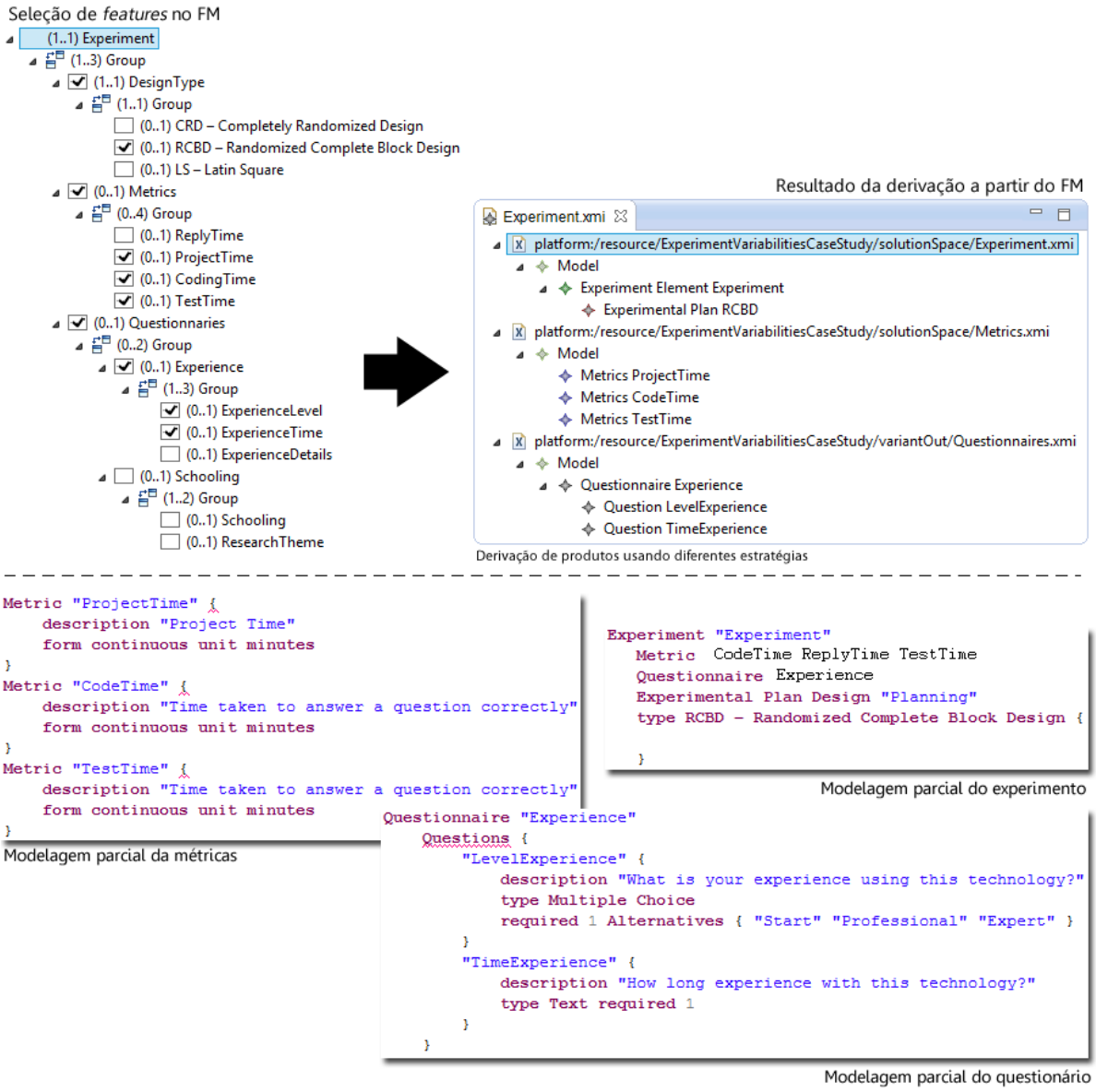


Figura 28. Derivação usando o *FeatureMapper:* (a) representação em arquivo XMI; (b) representação textual.

O resultado da derivação, como previsto, resultou em uma modelagem parcial do experimento. A Figura 28.b apresenta o resultado em notação textual usando as DSLs e a Tabela 3 os ajustes realizados na modelagem complementar.

Tabela 3. Modelagem complementar realizada após a derivação parcial

	Estado da modelagem parcial gerada a partir da seleção de *features*	**Modelagem complementar realizada**
Processo	Não foi gerado neste passo por não ter sido representado no modelo de *feature*	Modelado inteiramente neste passo
Métricas	- Atributo `relate` (não informado) precisa apontar para um processo; - Atributo `details` (não informado) precisa apontar uma tarefa, atividade ou artefato do processo.	- Incluído processo relacionado em cada métrica modelada; - Informado as tarefas medidas por cada métrica modelada.
Questionários	- Atributo `type` (não informado) é obrigatório.	- Incluído o atributo `type` com valor `Pos`; - Incluído o atributo relates apontando para o processo criado
Experimento	- É obrigatório informar ao menos um atributo `Factor` (não informado) com seus respectivos `Level`.	- Incluído referência ao processo do experimento modelado; - Incluído fatores com seus respectivos níveis.

A modelagem de processo não foi gerada, pois não foram incluídas variabilidades relativas a ela no modelo de *features*. A modelagem do experimento trouxe já definida as métricas e questionários relacionados e o tipo de design do experimento, mas ainda foi necessário especificar os fatores com seus respectivos níveis e o processo criado neste passo. As demais modelagens de métricas e questionários trouxeram resultados mais próximos da modelagem final, faltando apenas referenciar o processo relacionado e indicar o tipo de aplicação (no caso do questionário) e as tarefas a serem medidas (no caso das métricas). Após esses ajustes, a modelagem foi finalizada e transformada usando as mesmas transformações utilizadas na estratégia anterior para a geração dos *workflows* finais.

4.6. Sumários dos Resultados

Esta seção apresenta um sumário dos principais artefatos produzidos durante o estudo exploratório, agrupando-os pelas fases do método e suas respectivas atividades. A Tabela 4 sumariza os principais resultados em termos de artefatos produzidos durante a engenharia de domínio que resultou no desenvolvimento da abordagem generativa. A primeira coluna da tabela indica a atividade executada e na seguinte são listados os artefatos produzidos ao final da execução da respectiva atividade, com comentários acerca dos achados obtidos durante a sua execução ou detalhes do projeto/implementação.

Tabela 4. Sumário dos artefatos produzidos durante a Engenharia de Domínio

Atividade executada	Artefatos produzidos	
Separar domínios	• Relação de subdomínios do domínio de ESE	*Features* foram agrupadas em 4 grupos: experimento, processo, métrica e questionário
Identificar sobreposições	• Relação de sobreposições entre os subdomínios	Oito referências foram identificadas, sendo 4 simples e 4 com restrições adicionais
Implementar DSls individualmente	• Meta-modelos *EMF* e gramáticas em *xText* das DSLs	O xText foi usado para construir as DSLs, cujos meta-modelos estão no Apêndice A deste documento e as gramáticas na Seção 4.6
Especificar composição	• Meta-modelos compostos • Métodos adicionais em Java implementados para validar as referências com restrições adicionais	Meta-modelos foram compostos por meio de referências explícitas e métodos de validações foram implementados para os casos de referências com restrições adicionais
Implementar transformações	• Especificação em QVTo das transformações M2M	Foram aproveitados os modelos de transformações da própria abordagem generativa que utilizada linguagem QVTo

O resultado final da engenharia de domínio foi o desenvolvimento de uma abordagem generativa completa formada por um conjunto de DSLs, implementadas para modelar diferentes aspectos de um experimento controlado. Transformações M2M também foram especificadas e preparadas para transformar modelagens de experimentos realizadas utilizando as DSLs, durante a engenharia de aplicação, em arquivos de configuração a serem lidos por um motor de execução de *workflows*.

Durante a engenharia de aplicação, a abordagem generativa desenvolvida na fase anterior foi utilizada para derivar experimentos controlados, usando diferentes estratégias. A Tabela 5 sumariza os resultados finais produzidos pela engenharia de aplicação, independente da estratégia, distribuídos de forma análoga a anterior.

Tabela 5. Sumário dos artefatos produzidos durante a Engenharia de Aplicação

Atividade executada	Artefatos produzidos	
Modelar requisitos usando DSLs	• Modelagem individual de cada parte do produto	Irá variar a forma de como será produzida (manualmente ou automatizada) conforme estratégia adotada
Aplicar composição	• Especificação final do produto	Em alguns casos acontece em concomitância com a produção da atividade anterior, depois de validado
Executar as transformações	• Produto final	A transformação é processada por uma ferramenta de suporte a partir dos modelos especificados

Como o estudo estava interessado em avaliar a viabilidade da adoção das duas estratégias de derivação em uma abordagem generativa, realizamos a derivação de um mesmo produto duas vezes, aplicando em cada momento uma das estratégias e comparando qualitativamente os resultados produzidos por cada. Observamos que na derivação que combinou FMs e DSLs, foi necessário fornecer, além da especificação dos requisitos do produto que queríamos derivar, arquivos com as modelagens de outros experimentos com fragmentos passíveis de reuso. Além disso, um esforço extra inicial foi realizado para especificar um modelo de *features* que representasse as variabilidades existentes nessas modelagens e que pudesse ser usado pra reusar fragmentos dos modelos de forma automatizada. A seção seguinte irá discutir de forma mais aprofundada os principais achados da aplicação do método neste estudo exploratório.

4.7. Discussões dos Resultados

Concluída a aplicação do método, pudemos refletir sobre alguns resultados da sua aplicabilidade e viabilidade. De início, podemos afirmar que foi possível aplicá-lo em uma abordagem generativa que envolveu a utilização de múltiplas DSLs desenvolvidas com tecnologias dirigidas por modelos. Ademais, foi possível gerar produtos da LPS desenvolvida usando diferentes estratégias de derivação envolvendo composição de DSLs. Como resultado, destacamos alguns pontos para discussões, cujos principais achados estão apresentados na Tabela 6 e discutidos no decorrer desta seção.

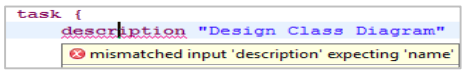
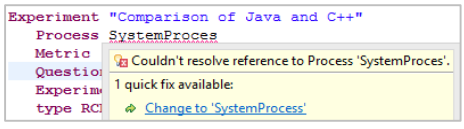
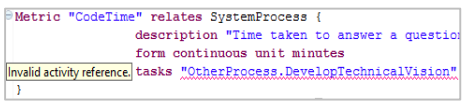
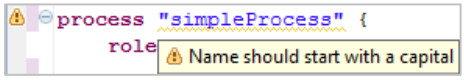
Tabela 6. Principais achados do estudo exploratório

Pontos para discussão	Principais achados
Utilização de tecnologias atuais de engenharia dirigida por modelos	✓ Avaliação dos recursos do xText para o desenvolvimento com composição de DSLs
Derivação de produtos utilizando diferentes estratégias	✓ Avaliação qualitativa da viabilidade das estratégias ✓ Uso de tecnologias distintas dirigidas por modelos combinadas para implementar tais estratégias ✓ Investigação do papel do FMs com múltiplas DSLs
Expressividade dos FMs e DSLs	✓ DSLs são mais expressivas que FMs ✓ DSLs podem ser usadas em complemento aos FMs
Impacto da separação de domínio em partes menores que se relacionam	✓ Subdomínios independentes e suas DSLs favorecem o reuso dos mesmos em outros cenários ✓ Favorece também o reuso de modelagens da LPS
Especificação de transformações	✓ O modo de implementação de transformações pode variar conforme cenário e requer planejamento

Utilização de tecnologias dirigidas por modelos. Os resultados mostram que é possível aplicar o nosso método em abordagens generativas com composição de DSLs utilizando tecnologias de engenharia dirigida por modelos. No estudo, as DSLs foram implementadas utilizando o framework *xText* que é baseado no meta-modelo *Ecore* do EMF. As referências entre as gramáticas foram implementadas no *xText* por meio de importação e referências explícitas entre os modelos, o que possibilitou ao *xText* gerenciar inconsistência entre os modelos automaticamente durante a etapa de modelagem. Recursos nativos do *xText*, tais como, assistentes inteligentes, também possibilitaram assistência à navegação e manutenção durante a modelagem e foram úteis para validar os modelos construídos. As restrições que envolvem a definição dos próprios artefatos dos modelos, tais como, a indicação de obrigatoriedade ou não de elementos, relações entre elementos em uma mesma gramática, ordem de definição e atributos, etc. foram validados pelos mesmos recursos a partir da especificação da própria gramática das DSLs. Apenas no caso de referências adicionais e de estilos que foi necessário implementar métodos adicionais para validar as restrições do domínio, utilizando linguagem Java.

A Tabela 7 apresenta um resumo da implementação que foi realizada utilizando o *xText* para implementar cada um dos tipos de restrições. A última coluna da tabela, apresenta ainda um exemplo do resultado da aplicação, com destaque a apresentação de guias de alertas, erros e sugestões fornecidas pelo *xText* e que apoiaram a gerência de consistência e manutenção dos modelos.

Tabela 7. Resumo do mecanismo de implementação adotado para cada tipo de restrição usando o *xText*

Tipo de restrição	Implementação com xText	Exemplo da aplicação
Artefatos individuais bem-definidos	A própria gramática da DSL impede que seja especificado um artefato com algum elemento/atributo obrigatório não informado	
Integridade entre artefatos	Utilizando referências explícitas nas gramáticas possibilitou apresentar alertas em caso de violação da integridade dos artefatos	
Referências com restrições adicionais	Combinando referências explícitas nas gramáticas e métodos de validações adicionais que implemente a restrição	
Restrições de estilo	Através de métodos de validação de implementem a restrição é possível validar sua violação	

Derivação de produtos usando diferentes estratégias. No estudo, derivamos produtos usando as duas estratégias previstas pelo método: utilizando apenas DSLs e combinando DSLs com FMs. Como resultado, observamos que a primeira delas demonstra ser mais apropriada para derivar produtos novos da LPS, cuja modelagem tivesse que ser construída do zero por não haver nenhum outro produto anterior derivado que pudesse, ao menos em parte, ter sua modelagem reutilizada. De modo inverso, a estratégia que combinou FMs com DSLs se apresenta como mais indicada quando já existem outros produtos similares derivados, com fragmentos de modelagens que pudessem ser reutilizados, e pouco recomendada para derivar produtos que não reutilizam nenhum fragmento de modelagens anteriores. Nesta segunda estratégia, o papel do FM foi selecionar partes da modelagem que se desejava reutilizar no novo experimento a ser derivado, enquanto na anterior o FM teve função apenas de rastreabilidade dos artefatos da LPS. Não foi avaliado o esforço gasto na aplicação de cada estratégia, porém, observamos que a estratégia combinando FMs e DSLs requer um esforço adicional de preparação do ambiente de derivação. Esse esforço, contudo, se torna desnecessário a partir do segundo produto derivado. Por outro lado, a modelagem com FMs e DSLs resultou em uma modelagem parcial que precisa de ajustes complementares (finalização da modelagem), enquanto na primeira estratégia já modelamos o produto final desde o início do processo. A Tabela 8 apresenta um resumo dos principais achados qualitativos do comportamento de cada uma das estratégias em diferentes situações: (i) reuso de modelos; (ii) derivação de produtos com muitas características diferentes dos que já foram derivados (ou quando não há nenhum produto derivado na abordagem ainda); (iii) derivação de produtos similares a outros já derivados; e o (iv) esforço extra necessário para executar a estratégia, especialmente em sua primeira aplicação ou em casos de evolução da LPS.

Tabela 8. Comportamento das duas estratégias de derivação em cada cenário

Cenário/Estratégia	**Apenas DSLs**	**DSLs com FMs**
Reuso de fragmentos de modelagens	Manual (*copy-paste*)	Automatizado (seleção de *features*)
Derivação de produtos diferenciados (quando não há fragmentos para serem reusados)	Mais indicada	Menos indicada
Derivações de produtos similares (com muitas características similares a outros já derivados)	Menos indicado	Mais indicado
Esforço extra despendido (na primeira execução da estratégia)	Nenhum	Especificação de um FM que represente as variabilidades dos fragmentos

Expressividade do FM e DSLs. Os problemas da expressividade dos FMs também foram investigados neste estudo. DSLs foram utilizadas com objetivo de solucionar tais problemas. Primeiramente, foram investigados os requisitos da abordagem generativa que o FM não foi capaz de expressar e, em seguida, adicionados tais requisitos às gramáticas das DSLs. As modelagens construídas para derivar produtos da LPS mostraram que tais requisitos podem ser expressos a partir das gramáticas implementadas, aplicando assim os limites de expressividade da abordagem em relação à abordagem generativa tradicional, orientada à *features*.

Impacto da separação em subdomínios no reuso de artefatos da LPS. A separação do domínio de ESE em subdomínios independentes e que se relacionavam, durante a fase de projeto, possibilitou a implementação de DSLs que modelavam perspectivas específicas do domínio, identificadas no estudo. Além disso, a separação em partes menores possibilita que algumas delas posam ser reaproveitadas em outros domínios. Este é o caso da linguagem de processos, que poderia ser reutilizada para especificar processos de desenvolvimento de software ou processos de negócios, além dos processos experimentais especificados no estudo. De mesmo modo, as demais linguagens poderiam ser reutilizadas, em outros domínios ou no mesmo para modelar novos produtos, obedecendo às dependências existentes (referências). No que se refere ao reuso de fragmentos de artefatos modelados, a estratégia de derivação que combinou FMs e DSLs tornou possível o reaproveitamento de partes da modelagem de um produto para outro, a partir da seleção de fragmentos reusáveis.

Tipos de estratégias de transformações. Observamos que durante a implementação das transformações, diferentes estratégias de execução dessas podem ser planejadas, a depender das DSLs envolvidas. Em geral, quando trabalhávamos com uma única DSL, antes de aplicar o método na abordagem de experimentos, uma única transformação era utilizada para geração dos arquivos de configuração dos *workflows*. Porém, com múltiplas DSLs, outras estratégias podem estar disponíveis. Em particular, neste trabalho, foram observadas as seguintes estratégias: (i) integrar as modelagens a serem transformadas em único modelo e aproveitar a especificação de transformação existente; ou (ii) especificar várias transformações, para cada DSL, gerando artefatos separados. Porém, os resultados ainda não são conclusivos o suficiente para apontar a alternativa mais vantajosa.

Estratégias para Implementação da Composição. Neste estudo, foi adotada a estratégia de referências explícitas para implementar as ligações existentes entre os elementos das DSLs, utilizando os recursos do *xText*. Contudo, a literatura prevê ainda a existência de outras alternativas para realizar a composição de modelos, as quais merecem ser exploradas em trabalhos futuros. Exemplos de tais alternativas de composição são: (i) *referências baseadas em nome* – proposta por (WARMER e KLEPPE, 2006), sugere conectar modelos referenciando seus elementos pelo seu nome. A desvantagem disto é que toda a semântica dos relacionamentos é transferida para o gerador de código, ou seja, a integração real acontece no nível de código, não ficando visíveis tais ligações no nível do modelo; (ii) *weaving models* – esta abordagem, apresentada em (GUILLAUME, FABRO, *et al.*, 2005), utiliza modelos de junção extensíveis para definir e visualizar correspondências entre os elementos em diferentes modelos, mantendo assim a implementação das referências fora dos modelos. Contudo, os modelos de junção só podem expressar a semântica das conexões, não representando explicitamente uma linguagem de construção. Além disso, o modelo de junção geralmente é destinado a ligar pares de modelos e, assim, não é adequado para a integração de um grande número de DSLs; (iii) *mapeamento de modelos* – abordagem apresentada em (BÉZIVIN, BÜTTNER, *et al.*, 2006), compreende que as transformações de modelos implementadas para mapear dois modelos distintos representam modelos próprios de mapeamento, que descrevem a relação entre os elementos dos modelos de entrada e saída, permitindo assim validar restrições de consistência existentes entre os modelos. No entanto, a especificação do mapeamento expressa a semântica de diferentes elementos da DSL apenas implicitamente, também não atribuindo um significado semântico explícito para cada modelo envolvido na transformação; e (iv) *ontologias* – por fim, em (BRÄUER e LOCHMANN, 2007) é previsto que a integração de linguagens deve ser baseada em fundamentos ontológicos, guiados por uma ontologia superior comum para linguagens de modelagem de software. Isto permite a reutilização de relações fundamentais, restrições e axiomas ao integrar uma nova DSL. A abordagem procura abordar as insuficiências das abordagens mencionadas anteriormente usando um conector modelo semântico. Apenas os elementos necessários para uma conexão semântica entre modelos participantes são mapeados para ontologia no conector semântica.

4.8. Conclusão

Este capítulo apresentou um estudo exploratório de investigação de mecanismos de integração entre modelos de *features* e linguagens específicas de domínio, utilizando tecnologias atuais de engenharia dirigidas por modelos e aplicando o método proposto neste trabalho, com o objetivo de responder a segunda questão de pesquisa desta dissertação, que questionava “como implementar abordagens generativas que envolvam a composição de múltiplas DSLs usando tecnologias atuais de engenharia dirigida por modelos”. Duas subquestões estavam envolvidas neste questionamento, a primeira busca investigar “como a composição de DSLs pode ser especificada e implementada durante a engenharia de domínio” e a segunda “como estratégias de derivação de produtos/sistemas de abordagens generativas que envolvam a composição de DSLs podem ser implementadas na engenharia de aplicação”.

Com os resultados do estudo, pudemos explorar a utilização de tecnologias que estão em evidência atualmente na acadêmica e na indústria, e que são baseadas em engenharia dirigida por modelos, para implementar sintaxes de DSLs, especificando os pontos de sobreposições existentes entre elas, e assim responder aos questionamentos mencionados. Além disso, uma linguagem para transformações de modelos pode ser utilizada para implementar as transformações necessárias para geração dos artefatos finais da abordagem. Especificamente, dois *frameworks* foram utilizados e avaliados neste estudo, o *xText*, baseado no *Eclipse Modeling Framework* (EMF), e o *FeatureMapper*. O *xText* já apresenta nativamente uma série de recursos que podem ser aproveitados para desenvolvimento com composição de DSLs. Já o *FeatureMapper* é útil para conectar *features* de um modelo de *features* com fragmentos de modelagens construídas usando as DSLs do *xText*, automatizando assim o reuso de modelagens na execução da estratégia de derivação que combina FMs e DSLs. Como previsto nos estudos relacionados, os resultados do nosso estudo mostraram também que é possível derivar produtos em abordagens generativas com composição de DSLs, utilizando apenas DSLs ou combinando-as com FMs. O uso de DSLs é útil para ampliar os limites de expressividade dos FMs e ainda fornecer ao usuário final novas experiências de interação (*wizard*, arquivos de configuração, modelos, etc.). DSLs podem ainda ser implementadas por tecnologias atuais de engenharia dirigida por modelos, tal como

xText, e mapeadas à *features* em um FM, usando *frameworks* de mapeamento de variabilidades com modelos, tais como o *FeatureMapper*, ambos utilizados e avaliados neste estudo.

Além disso, com a aplicação do método, foi possível observar que a separação de um domínio em partes menores favorece reuso dos artefatos modelados na abordagem generativa. Desse modo, identificar subdomínios, separando-os e identificando sobreposições entre eles, ainda na fase de projeto de domínio é de fundamental importância para permitir a especificação separada de *features* pertencentes a diferentes domínios do sistema, assim como o reuso futuro das DSLs em outras abordagens generativas. Foi possível também observar que os problemas reconhecidos pela literatura envolvendo composição de DSLs, tais como, as violações às restrições dos modelos e gerenciamento de inconsistências, foram identificados e devidamente tratados pelo método, e assim contempladas soluções para os mesmos durante o estudo. As DSLs implementadas também podem ser reutilizadas em outros domínios e os fragmentos das modelagens baseadas nas linguagens específicas de domínio.

Por fim, como todo estudo empírico, este também apresentou algumas ameaças à sua validade que precisam ser apresentadas e discutidas (Seção 4.8.1). Por outro lado, o estudo também trouxe importantes contribuições, que estão expostas na Seção 4.8.2.

4.8.1. Ameaças à Validade e Limitações do Estudo

Estudos empíricos, de modo geral, estão sujeitos às ameaças. Assim, não diferente, o mesmo ocorre com o estudo exploratório realizado nesta pesquisa. Todavia, identificá-las previamente pode minimizar o seu efeito. Em alguns casos tais ameaças geram limitações do estudo que precisam ser de mesmo modo observadas para serem corrigidas em trabalhos futuros. Especificamente neste estudo, as ameaças principais que podem limitá-lo são:

- A *quantidade de domínios envolvidos na investigação* é uma das limitações do estudo exploratório. No estudo, o método foi aplicado a apenas um domínio, o que pode ter limitado os resultados obtidos. Tal fato impediu que

pudéssemos analisar o comportamento do método para diferentes domínios, traçando comparações entre as aplicações para que pudesse ser possível delimitar qual o foco específico de atuação do método, ou seja, os domínios que melhor são contemplados pelas atividades do método. Em estudos futuros, certamente, estaremos interessados em avaliar novos domínios, diferentes do que foi avaliado neste estudo.

- A *aplicação do método pelos próprios pesquisadores* que o projetaram é outra limitação do estudo que pode ter causado algum efeito de aprendizado. Para lidar com tal limitação, a ideia seria conduzir experimentos controlados com a participação explicita de *subjects* que não fizessem parte do método proposto, mas por limitação de pessoal e tempo para conclusão da pesquisa, não foi realizada neste estudo.

- O *uso de tecnologias específicas* para aplicação do método trouxe-nos resultados de implementação da abordagem também específicos para tais tecnologias utilizadas, não podendo ser expandidos para outras tecnologias sem prévia análise das mesmas.

4.8.2. Principais Contribuições do Estudo

As principais contribuições deste estudo foram: (i) a avaliação de um método proposto para abordagens generativas com composição de DSLs; (ii) a investigação dos problemas e soluções da integração de DSLs com FMs; e (iii) o desenvolvimento de uma abordagem generativa de experimentos controlados usando composição de DSLs.

5. TRABALHOS RELACIONADOS

Este capítulo apresenta os trabalhos relacionados a esta dissertação, organizados em duas áreas principais: composição de DSLs e desenvolvimento de LPS. Comparações entre os trabalhos e o nosso método também são apresentadas.

5.1. Abordagens para Composição de DSLs

Em (HESSELLUND, CZARNECKI e WASOWSKI, 2007) são relatados os resultados de um estudo de caso que envolveu a realização da composição de DSLs baseadas em XML de um framework o*pen source* para desenvolvimento de aplicações empresariais, o Apache Open for Business (OFBiz). O estudo utilizou a ferramenta SmartEMF, proposta anteriormente por um dos autores (HESSELLUND, 2007) e que estende o meta-modelo EMF, utilizando a linguagem de programação *Prolog*, para tratar/validar restrições de modelos. O estudo também investigou alguns dos principais problemas (e suas respectivas soluções) envolvendo restrições à consistência de modelos durante o desenvolvimento com múltiplas DSLs, mas limitou seus resultados por ter investigado apenas cenários com DSLs baseadas em XML e não ter proposto nenhuma sistematização para aplicação das soluções encontradas. Assim, como continuidade dessa pesquisa, seus achados foram aprimorados em (HESSELLUND e LOCHMANN, 2009), que propôs um método voltado ao desenvolvimento com composição de DSLs. Os resultados desses dois trabalhos exerceram forte influência sobre a nossa pesquisa. Suas contribuições serviram de base para formulação do nosso método, que incluiu algumas de suas soluções para lidar com composição de DSLs no desenvolvimento generativo, conforme já citada no capítulo 3 desta dissertação.

Outros pesquisadores (LOCHMANN e BRÄUER, 2007; LEDCZI, NORDSTROM, *et al.*, 2011; HAAN, 2008) também investigaram problemas envolvendo composição de DSLs, porém poucos deles propuseram metodologias focadas no desenvolvimento generativo com múltiplas DSLs. Warmer e Kleppe (WARMER e KLEPPE, 2006) propuseram uma abordagem para desenvolvimento de fábricas de software (*software factories*) utilizando múltiplas DSLs em conjunto com técnicas de desenvolvimento dirigido por modelos, tal como transformações modelo-

para-modelo e modelo-para-código. A abordagem é baseada no uso da ferramenta Microsoft DSL, mas pode ser estendida para outros ambientes. Eles seguiram o processo de desenvolvimento de fábrica de software dirigido por modelos definido por (GREENFIELD, SHORT, *et al.*, 2004), que já utiliza DSLs e transformações de modelos, e aplicaram no desenvolvimento de uma fábrica de software baseada na web, a *SMART-Microsoft Software Factory*. Durante a aplicação do processo, o domínio da aplicação foi organizado em partes menores que eles chamaram de modelos parciais (*partial models*). Cada parte endereça um problema específico do domínio com DSLs que se referenciam. A abordagem sugere conectar modelos referenciando elementos do modelo pelo nome a partir de outros modelos, conhecida como referências baseadas em nome. Contudo, essa solução apresenta uma desvantagem, por transferir toda a semântica para o gerador de código, ou seja, a integração real dos modelos acontece no nível de código (HAAN, 2008). Neste caso, no nível do meta-modelo, não é possível visualizar referências entre os elementos, não podendo assim reutilizá-las em outras aplicações. Em nosso método, há a previsão de realização de atividades de identificação de referências entre os modelos e codificação das mesmas já no nível de meta-modelo, ainda durante o desenvolvimento da abordagem generativa. Neste caso, diferentemente de Warmer e Kleppe, nós utilizamos referências explícitas para implementar as referências existentes entre os meta-modelos das nossas DSLs.

Alguns outros estudos relacionados ao tema de composição de DSLs foram realizados. Haan (HAAN, 2008) menciona dois possíveis tipos de composição presentes na literatura e soluções encontradas no espaço solução: (i) modelagem multi-hierárquica, onde composições hierárquicas de estilos de modelagem distintas são combinadas para tirar partido das capacidades únicas e expressividade dos estilos de modelagem distintas; e (ii) modelagem *multi-view*, onde modelos distintos e separados do mesmo sistema são construídos para representar os diferentes aspectos do sistema. Bräuer e Lochmann (BRÄUER e LOCHMANN, 2007) propõem uma abordagem para integrar múltiplas DSLs durante o desenvolvimento de sistemas. Eles se concentraram na fase de projeto e propuseram a utilização de técnicas de ontologias para abordar relacionamentos semânticos entre elementos individuais dos modelos. O uso de conectores semânticos permite armazenar um conjunto maior de informações sobre os elementos de diversos modelos a partir de

relacionamento. Um estudo de caso trivial foi realizado no trabalho para ilustrar a aplicação da abordagem. Contudo, diferente do nosso trabalho, os autores também não apresentam uma metodologia com um encadeamento de atividades que possibilite aplicar sistematicamente a abordagem deles no desenvolvimento de abordagens generativas. Nenhum dos trabalhos encontrados propôs um método para desenvolvimento de abordagens generativas com múltiplas DSLs. Por outro lado, como é apresentado na Seção 5.2, abordagens para desenvolvimento generativo foram propostas, mas sem considerar soluções para múltiplas DSLs. Neste sentido, nosso trabalho, visando complementar esta lacuna, foi além dos trabalhos citados e propôs um método que define um conjunto de atividades a serem realizadas por papéis definidos, de forma encadeada e sistematizada, para o desenvolvimento de abordagens generativas com múltiplas DSLs.

5.2. Abordagens para Desenvolvimento de LPS

Esta seção apresenta dois trabalhos relacionados que propuseram uma abordagem para desenvolvimento de LPS, citando o uso de DSLs, porém sem abordar composição. Além de apresentar os trabalhos, a seção realiza também uma breve comparação com a nossa.

5.2.1. Abordagem TENTE

Em (FUENTES, NEBRERA e SÁNCHEZ, 2009) é proposto uma abordagem, conhecida como TENTE[3], para LPSs dirigidas a produtos, que gerenciam um número finito de produtos, cada qual comercializado em larga escala para diversos clientes (ZSCHALER, SÁNCHEZ, *et al.*, 2011). A abordagem TENTE consiste em um processo dirigido por modelos e orientado à *features* focado nas etapas de projeto e implementação, tanto na engenharia de domínio como de aplicação. Técnicas MDD são utilizadas para automatizar, parcialmente, o processo de engenharia de domínio e, completamente, a engenharia de aplicação. Modelos arquiteturais são expressos em UML 2.0 (http://uml.org/) e implementados com a

[3] TENTE é o nome espanhol para Lego. Este nome se dá pelo fato da abordagem visualizar uma LPS como um jogo de Lego, onde produtos são especificados pela junção de blocos pré-construídos.

linguagem CaesarJ (ARACIC, GASIUNAS, *et al.*, 2006), uma linguagem baseada em Java com características específicas destinadas à programação orientada à *features*, provendo mecanismos, tais como, *virtual classes* e *mixin composition* (GASIUNAS, NÚÑEZ, *et al.*, 2011). Essa linguagem permite implementar cada *feature* em módulos separados que podem ser compostos usando mecanismo de modularização próprios para gerar um produto desejado.

O processo da abordagem TENTE compreende cinco etapas, tal como representado na Figura 29. As primeiras três etapas correspondem à engenharia de domínio e fomentam a criação da infraestrutura para geração de produtos, os quais são derivados nas duas últimas etapas, relacionadas à engenharia de aplicação. As etapas do processo estão descritas em mais detalhes a seguir.

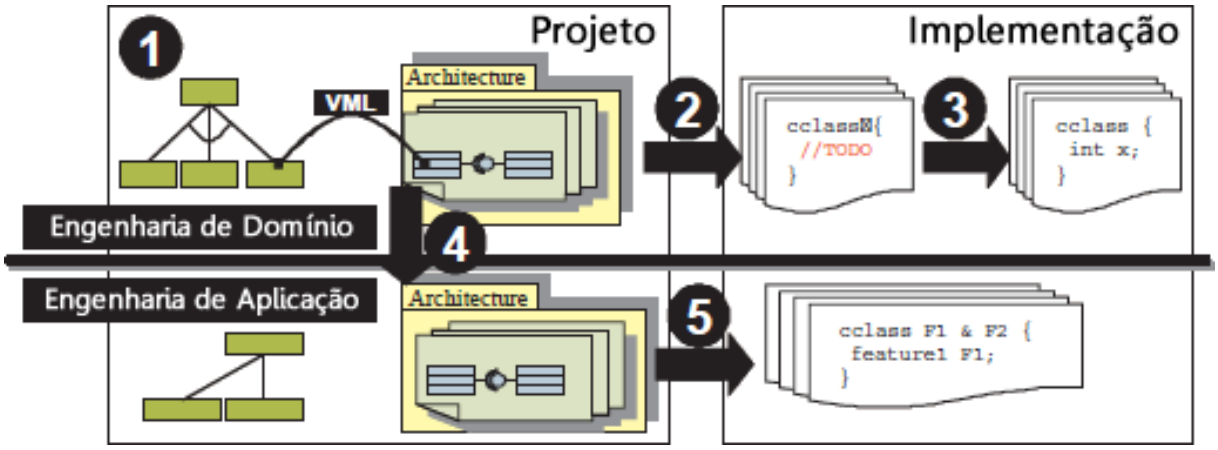


Figura 29. Visão Geral do Processo TENTE – Adaptada de *(FUENTES, NEBRERA e SÁNCHEZ, 2009)*

Passo 1: Projeto arquitetural

O primeiro passo é a especificação de um modelo arquitetural para LPS. O modelo arquitetural é composto por três elementos: (i) modelo de *features* baseado em cardinalidade; (ii) um modelo UML 2.0; e (iii) uma especificação usando uma Linguagem de Modelagem de Variabilidade (*Variability Modelling Language* – VML). Modelos de *features* baseados em cardinalidade (KRZYSZTOF, HELSEN e EISENECKER, 2005) especifica quais partes da arquitetura são variáveis e se elas são alternativas mutuamente exclusivas ou simplesmente *features* opcionais. Eles devem ser derivados da especificação de requisitos da LPS recebidos como entrada pela abordagem. Se forem utilizadas para especificar variabilidades dos requisitos, esta informação precisará ser convertida em FMs baseados em cardinalidade. O segundo elemento, os modelos UML 2.0, é usado para representar a arquitetura de

referência da LPS e deve conter todos os componentes projetados, incluindo os artefatos comuns e variáveis. Para ligar as *features* do FM à arquitetura de referência, uma conexão usando uma VML é especificada. Uma VML é uma DSL que funciona como um modelo de configuração a conectar explicitamente as variabilidades do espaço problema (modelo de *features*) com o espaço solução (modelos UML 2.0). Há diversas abordagens de VML na literatura, tais como a VML4Arch (LOUGHRAN, SÁNCHEZ, *et al.*, 2008; SÁNCHEZ, LOUGHRAN, *et al.*, 2008), utilizada nesta abordagem; e a *FeatureMapper* (HEIDENREICH, KOPCSEK e WENDE, 2008), que se baseia em modelos EMF e foi utilizada no nosso estudo. Uma especificação usando uma VML pode ser automaticamente traduzida em transformações de modelos para gerar implementações finais ou parciais em uma linguagem qualquer de propósito-geral.

Passo 2: Transformação dos modelos arquiteturais em implementação

A partir da arquitetura de referência, usando um gerador de código, parte da implementação é automaticamente gerada (Figura 29 – Passo 2). Mais especificamente, é gerada a estrutura lógica dos componentes/classes com as conexões existentes entre eles. A implementação de métodos comportamentais é realizada manualmente pelos desenvolvedores da LPS no passo seguinte.

Passo 3: Implementação da engenharia de domínio

A implementação parcialmente gerada no passo anterior é então finalizada neste passo com inserção das devidas lógicas de negócios aos componentes (Figura 29 – Passo 3). O resultado desta fase é um conjunto de componentes que implementam todos os requisitos levantados para a LPS. Os componentes são mantidos separados em pacotes (definição oriunda dos modelos UML) de modo que possam ser instanciados e devidamente conectados para gerar produtos específicos durante a engenharia de aplicação. Este passo encerra a engenharia de domínio na abordagem.

Passo 4: Derivação de um específico modelo arquitetural

Nesta etapa é gerada uma instância do modelo de *features* que define uma configuração válida de *features*, selecionadas para serem incluídas em um produto específico (Figura 29 – Passo 4). A partir dessa configuração o modelo arquitetural

desejado é automaticamente gerado, usando a especificação VML. A especificação VML é compilada para gerar as transformações de modelos que derivarão a arquitetura específica automaticamente, sem a necessidade de modelar as transformações manualmente.

Passo 5: Derivação de uma específica implementação

Por fim, o modelo de arquitetura gerado no passo anterior é automaticamente transformado, usando um gerador de código que resultará em uma implementação completa do produto. A implementação gerada utiliza a linguagem CaesarJ que permite que os componentes derivados já venham devidamente instanciados e conectados para montar o produto final desejado.

5.2.2. Abordagem MAPLE

A abordagem MAPLE (*Model-driven Aspect-oriented Product Line Engineering*) integra elementos da engenharia dirigida por modelos com técnicas do desenvolvimento orientado a aspectos para facilitar a modelagem e implementação variabilidades em uma linha de produto, especialmente em LPS voltadas à solução. LPSs dirigidas à solução gerenciam um número indeterminado de produtos, onde, em geral, cada produto é gerado para solucionar um problema específico do domínio (GROHER, FIEGE, *et al.*, 2011). Na abordagem MAPLE, modelos são construídos para descrever as variabilidades da LPS e transformações de modelos, assim como técnicas de orientação à aspectos, são utilizadas para gerar os produtos da LPS. A Figura 30 apresenta uma visão da abordagem.

A partir da especificação dos requisitos do domínio é desenvolvido um meta-modelo que compreende as variabilidades da LPS, e define o vocabulário e gramática usada para desenvolver modelos no espaço de problema. Ainda no espaço de problema, de acordo com os requisitos do produto desejado, é criado um modelo-instância do meta-modelo da LPS, usando uma DSL. Uma ferramenta-editor deve ser usada para suportar a criação de modelos utilizando DSLs. O uso de DSL permite a criação de um número infinito/indeterminado de modelos válidos, contemplando os diferentes tipos de variabilidades previstos pelo domínio. No espaço de solução, um meta-modelo arquitetural que implemente os requisitos na

LPS é criado (ou usado um meta-modelo existente) e mapeado formalmente com o meta-modelo do espaço problema, de modo a possibilitar a execução de transformações.

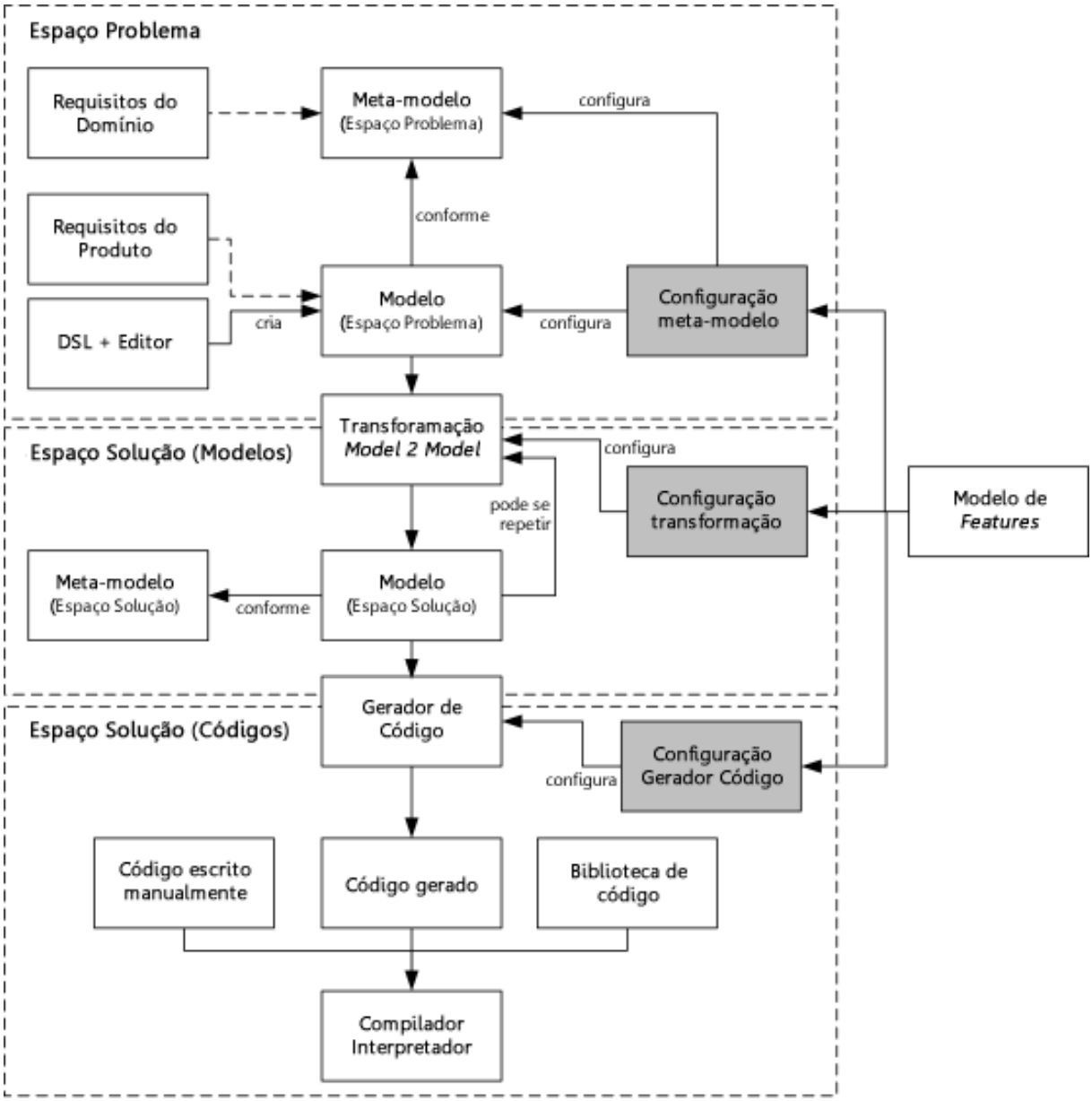


Figura 30. Visão geral da abordagem MAPLE – Adaptada de *(GROHER, FIEGE,* et al.*, 2011)*

5.2.3. Comparação entre as Abordagens

Das duas abordagens apresentadas, a MAPLE é a que mais se assemelha ao nosso método, especialmente pelo fato de promover a especificação de DSLs para amparar o desenvolvimento de LPSs. A abordagem MAPLE propõe a especificação de DSLs para serem utilizadas durante as diferentes fases do desenvolvimento de

um software, permitindo assim usá-las para especificar um incontável número de produtos em uma LPS, usando ferramentas de suporte e modelos de transformações. Contudo, a abordagem não apresenta mecanismos para identificar sobreposições entre domínios ou gerenciar consistências entre modelos, típico de um desenvolvimento com composição de DSLs, que exploramos em nosso método.

Do outro lado, a abordagem TENTE sugere um desenvolvimento mais convencional, utilizando linguagens de propósito-geral. Com isso, exalta vantagens não alcançadas no desenvolvimento com DSLs, tal como, o custo reduzido para o desenvolvimento e manutenção da LPS, especialmente quando relacionada ao esforço requerido para desenvolvimento e manutenção de arquiteturas de LPSs envolvimento múltiplas DSLs, específicos modelos de transformação e geradores de código. Tal fato requer um maior suporte ferramental e a utilização de abordagens baseadas em DSLs para orientar o desenvolvimento.

Em contrapartida, o uso de abordagens genéricas não possibilita focar em um domínio em particular, deixando de lado características específicas do domínio em utilização, podendo gerar assim fortes problemas de expressividade da linguagem para alguns domínios. Criar configurações de um modelo de *features* não é uma tarefa complexa, porém pode não contemplar plenamente características particulares de alguns domínios. Diferentemente, DSLs, apesar de maior o custo do desenvolvimento e elevar a possibilidade de problemas de inconsistências, permite a realização de modelagens mais expressivas e próximas da necessidade do usuário.

6. CONCLUSÃO

Este capítulo apresenta as conclusões desta dissertação de mestrado. A Seção 6.1 apresenta os resultados alcançados para cada questão de pesquisa. A Seção 6.2 discute as principais limitações do trabalho. A Seção 6.3 ressalta suas principais contribuições. Finalmente, a Seção 6.4 apresenta proposições de trabalhos futuros que estendam e continuem os avanços desta pesquisa.

6.1. Análise dos Resultados da Dissertação

A primeira questão de pesquisa (QP1) — "*De que forma modelos de features e linguagens específicas de domínio podem ser usados de forma integrada durante o desenvolvimento de abordagens generativas?*" — estava mais inclinada a investigar e propor soluções que promovam a integração de FMs e DSLs durante o desenvolvimento generativo. Duas questões específicas relacionadas à QP1 foram exploradas nesta dissertação. Para respondê-las, nos baseamos em três abordagens da literatura para propor um método que sistematiza a integração de FMs e DSLs durante o desenvolvimento de abordagens generativas (capítulo 3). As abordagens investigadas serviram de base para a proposição deste trabalho, que se baseou em uma abordagem de integração de FM e DSLs (VOELTER e VISSER, 2011) para incorporar à abordagem generativa do (CZARNECKI e EISENECKER, 2000) as atividades de composição de DSLs propostas pelo (HESSELLUND, 2009). Como resultado, nosso método sistematiza a junção de metodologias existentes para o desenvolvimento generativo com composição de múltiplas DSLs.

Assim, foi possível explorar e responder às duas questões de pesquisa específicas relacionadas à QP1: (i) "*Qual o papel e utilidade do modelo de features em uma abordagem generativa que envolva a composição de múltiplas linguagens específicas de domínio durante a engenharia de domínio?"* — foi respondida pelo método proposto ao utilizar o FM como artefato de entrada e base para identificação dos domínios existentes em uma LPS. O FM também foi utilizado para reutilizar fragmentos de DSLs durante o processo de derivação e para fins de rastreabilidade da LPS durante um processo de evolução ou manutenção da mesma; e (ii) *"Que estratégias de derivação podem ser utilizadas para abordagens generativas que*

envolvam a integração do modelo de features e múltiplas linguagens específicas de domínio? Quais as vantagens e desvantagens de tais estratégias?" — foi respondida ao incorporarmos ao método, duas estratégias de derivação de produtos, uma utilizando apenas DSLs e outra combinando DSLs com FMs. A primeira estratégia é mais indicada e, portanto, mais vantajosa, para derivar produtos com requisitos novos, ainda não incluídos em nenhum outro processo de derivação de produtos, assim não tendo fragmentos de modelagens anteriores para reutilizar. Já a segunda estratégia, requer um esforço extra inicial de especificar um FM que represente as variabilidades dos artefatos de modelagem da LPS, sendo mais indicada para derivar produtos que possuem requisitos similares a outros já derivados. Neste caso, o FM possibilitará um reuso automatizado da modelagem desses requisitos, necessitando apenas uma complementação manual para os novos requisitos.

A segunda questão de pesquisa (QP2), — *"Como implementar abordagens generativas que envolvam a composição de múltiplas linguagens específicas de domínio usando tecnologias atuais de engenharia dirigida por modelos?"* — uma vez levantadas soluções na QP1, estava mais interessada em como implementar tais soluções. Desse modo, para investigá-la, foi realizado um estudo exploratório que avaliou o método proposto para implementar uma abordagem generativa que envolve a modelagem de experimentos controlados e a geração de *workflows* que representam tais experimentos. As questões específicas do estudo exploratório buscaram responder a esta questão de pesquisa e os seus resultados foram apresentados no capítulo 4 deste documento. O estudo exploratório envolveu a definição de quatro DSLs que podem ser utilizadas para modelagem uma família de experimentos.

Com os resultados do estudo, pudemos extrair respostas para as questões de pesquisa específicas relacionada à QP2: (i) *"Como a composição de linguagens específicas de domínio pode ser especificada e implementada durante a engenharia de domínio?"* — foi respondida pelo estudo, uma vez que foram identificadas sobreposições entre as DSLs desenvolvidas na engenharia de domínio, as quais foram implementadas como composições durante a etapa de implementação de domínio. As DSLs foram desenvolvidas utilizando um *framework* baseado em modelos, o *xText,* que vem recebendo atenção tanto da comunidade científica quanto da indústria; e (ii) "*Como estratégias de derivação de produtos/sistemas de*

abordagens generativas que envolvam a composição de linguagens específicas de domínio podem ser implementadas na engenharia de aplicação?” — foi respondida de modo análogo pelo estudo, que utilizou as DSLs construídas com o *xText* na ED para modelar produtos a serem derivados durante a estratégia de derivação que utilizava apenas DSLs. As mesmas DSLs também foram usadas em conjunto com um FM modelado com a ferramenta *FeatureMapper*, também dirigida por modelos, para automatizar o reuso de modelos durante a execução da estratégia que combinou DSLs com FM. Por fim, especificações de transformações de modelo para modelo foram criadas usando a linguagem de transformação de modelos QVTo para gerar os arquivos de configuração finais (produto final da abordagem).

6.2. Limitações do Trabalho

Os resultados deste estudo forneceram evidências importantes para discutir e responder as questões de pesquisa desta dissertação. Contudo, existem limitações na pesquisa realizada que precisam ser destacadas para serem exploradas em trabalhos futuros. São essas:

- Foi realizada uma revisão do estado da arte para investigar a existência de outros trabalhos que lidem com desenvolvimento generativo com múltiplas DSLs, mas é importante a condução de uma revisão sistemática da literatura que busque ampliar o escopo dos trabalhos identificados nesta dissertação;
- O método proposto não explorou em detalhes as relações existentes entre os requisitos da LPS (especificados através de modelos de requisitos, modelos de *features* e/ou DSLs) com representações ou modelos da sua arquitetura. Este aspecto é fundamental, considerando que a arquitetura é um dos elementos centrais desenvolvido para uma abordagem generativa;
- Nosso estudo também observou que estratégias podem ser usadas para implementar as transformações em função da definição de diferentes DSLs ou mesmo da integração com o modelo de *features*. Entretanto, o método não incluiu atividade para identificar tais variações, nem a dissertação explorou em detalhes tal aspecto.

6.3. Principais Contribuições

A seguir destacamos algumas das principais contribuições desta dissertação:

- Levantamento e identificação de soluções para lidar com composição de linguagens específicas de domínio e abordagens generativas;
- Levantamento e identificação dos diferentes tipos de integração que podem ocorrer entre modelos de *features* e linguagens específicas de domínio durante o desenvolvimento generativo;
- Proposta de um Método para o Desenvolvimento de Abordagens Generativas com Composição de Múltiplas Linguagens Específicas de Domínio;
- Condução de um estudo exploratório para avaliação do método no domínio de experimentos controlados em engenharia de software experimental;
- Projeto e implementação de uma Abordagem Generativa para o domínio de experimentos controlados em engenharia de software experimental, baseado no uso das tecnologias EMF e *xText*.

6.4. Trabalhos Futuros

Como trabalhos futuros, planejamos estender os resultados desta pesquisa de modo a lidar com limitações existentes do trabalho, assim como ampliar nossas contribuições. Os seguintes trabalhos de pesquisa podem ser desenvolvidos como desdobramento desta dissertação:

- Realizar uma revisão sistemática da literatura para ampliar nosso escopo de trabalhos sobre ferramentas e metodologias voltadas ao desenvolvimento generativo com composição de DSLs. Com isso, podemos expandir nossas análises comparativas com outros trabalhos relacionados e também refinar nosso método com a inserção de novos elementos;
- Aplicar o método proposto em outros domínios para que possamos comparar o seu comportamento em diferentes cenários e contextos;
- Realizar estudos com participantes não envolvidos na criação do método, de modo a ampliar os resultados e análise da sua usabilidade;

- Conduzir estudos experimentais para comparar as diferentes estratégias de derivação previstas pelo método, para que possamos obter análises quantitativas dos resultados. Tais estudos podem comparar abordar diferentes aspectos, tais como, a facilidade para derivar determinados produtos, o grau de reuso com cada abordagem, etc.;
- Por fim, outra frente de trabalho viável será avaliar e comparar outros *frameworks*, que assim como *xText*, possam ser utilizados para oferecer suporte para o desenvolvimento com composição de múltiplas DSLs.

REFERÊNCIAS

ARACIC, I. et al. An Overview of CaesarJ. In: AWAIS, R.; MEHMET, A. **Transactions on Aspect-Oriented Software Development I**. Berlin: Springer-Verlag, 2006. p. 135-173.

BETTINI, L. **A DSL for writing type systems for Xtext languages**. Proceedings of the 9th International Conference on Principles and Practice of Programming in Java. New York: ACM. 2011. p. 31-40.

BÉZIVIN, ; JOUAULT, F. **Using ATL for Checking Models**. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT). Tallinn: [s.n.]. 2005. p. 69-81.

BÉZIVIN, J. et al. **Model Transformations? Transformation Models!** Proceeding MoDELS'06 Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems. Heidelberg: Springer-Verlag Berlin. 2006. p. 440-453.

BRÄUER, M.; LOCHMANN, H. **Towards Semantic Integration of Multiple Domain-Specific Languages Using Ontological Foundations**. Proceedings of 4th International Workshop on Software Language Engineering (ATEM) on MoDELS. Nashville: [s.n.]. 2007.

CAMPOS NETO, E. B. et al. **Composição de Linguagens de Modelagem Específicas de Domínio:** Um Estudo Exploratório. III Brazilian Workshop on Model-Driven Software Development. Natal: [s.n.]. 2012. p. 41-48.

CAMPOS NETO, E. B. et al. **Composition of Domain Specific Modeling Languages:** An Exploratory Study. Composition of Domain Specific Modeling Languages: An Exploratory Study. Barcelona: Springer. 2013.

CLEMENTS, P.; NORTHROP, L. **Software Product Lines:** Practices and Patterns. Professional. [S.l.]: Addison-Wesley, 2011.

CZARNECKI, K.; EISENECKER, U. **Generative Programming:** Methods, Tools, and Applications. New York: Addison-Wesley Professional, 2000.

CZARNECKI, K.; HELSEN, S. Feature-Based Survey of Model Transformation Approaches. **IBM Systems Journal: Model-driven software development**, Riverton, v. 45, n. 3, p. 621-645, Julho 2006. ISSN 0018-8670.

DEELSTRA, S.; SINNEMA, M.; BOSCH, J. Product derivation in software product families: a case study. **Journal of Systems and Software - Special issue: The new context for software engineering education and training**, New York, v. 74, n. 2, p. 173-194, 15 Janeiro 2005.

FOWLER, M.; PARSONS, R. **DSL:** Linguagens Específicas de Domínio. Tradução de Eduardo Kessler Piveta. Porto Alegre: Bookman, 2013.

FREIRE, M. A. et al. **A Model-Driven Approach to Specifying and Monitoring Controlled Experiments in Software Engineering**. 14th International Conferenc on Product-Focused Software Process Improvement (PROFES). Pafos: [s.n.]. 2013.

FREIRE, M. et al. **Automatic Deployment and Monitoring of Software Processes:** A Model-Driven Approach. Conference on Software Engineering and Knowledge Engineering. Miami/Florida: [s.n.]. 2011.

FREIRE, M. et al. **Software Process Monitoring using Statistical Process Control Integrated in Workflow Systems**. Conference in Software Engineering Knowledge Engineering. San Francisco/California - USA: [s.n.]. 2012.

FUENTES, L.; NEBRERA, C.; SÁNCHEZ, P. **Feature-Oriented Model-Driven Software Product Lines:** The TENTE Approach. Proceedings of the Forum of the 21st International Conference on Advanced Information Systems (CAiSE). Amsterdam: [s.n.]. 2009. p. 67-72.

GAMMA, E. et al. **Design patterns:** elements of reusable object-oriented software. [S.l.]: Addison-Wesley, 1994.

GASIUNAS, V. et al. Product Line Implementation with ECaesarJ. In: RASHID, A.; ROYER, J.-C.; RUMMLER, A. **Aspect-Oriented, Model-Driven Software Product Lines:** The AMPLE Way. New York: Cambridge University Press, 2011. Cap. 6, p. 161-196.

GREENFIELD, J. et al. **Software Factories:** Assembling Applications with Patterns, Models, Frameworks, and Tools. [S.l.]: Wiley, 2004. ISBN 978-0-471-20284-4.

GROHER, I. et al. Solution-driven software product line engineering. In: RASHID, A.; ROYER, J.-C.; RUMMLER, A. **Aspect-Oriented Model-Driven Software Product Lines:** The AMPLE WAY. New York: Cambridge University Press, 2011. Cap. 11, p. 316-344.

GUILLAUME, G. et al. **AMW:** A Generic Model Weaver. Premières Journées sur l'Ingénierie Dirigée par les Modèles. [S.l.]: [s.n.]. 2005.

HAAN, J. D. Domain-Specific Modeling needs multi-models. **The Enterprise Architect:** Building an agile enterprise, 03 Novembro 2008. Disponivel em: <http://www.theenterprisearchitect.eu/archive/2008/11/03/domain-specific-modeling-needs-multi-models>. Acesso em: 29 Maio 2013.

HEIDENREICH, F.; KOPCSEK, J.; WENDE, C. FeatureMapper: Mapping Features to Models. **Proceedings of the 30th International Conference onf Software Engineering**, Leipzig, 10-18 Maio 2008. 943-944.

HESSELLUND , A.; LOCHMANN, H. **An Integrated View on Modeling with Multiple Domain-Specific Languages**. Proceedings of the IASTED International Conference Software Engineering (SE 2009). [S.l.]: [s.n.]. 2009. p. 1-10.

HESSELLUND, A. **SmartEMF:** guidance in modeling tools. 22nd ACM SIGPLAN Conference on OOPSLA. New York: [s.n.]. 2007. p. 945-946.

HESSELLUND, A. **Domain-specific multimodeling**. IT University of Copenhagen. Denmark. 2009.

HESSELLUND, A.; CZARNECKI, K.; WASOWSKI, A. **Guided Development with Multiple Domain-Specific Languages**. ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007). Nashville: Springer. 13 Setembro 2007. p. 46-60.

JEDLITSCHKA, A. et al. **Relevant Information Sources for Successful Technology Transfer:** A Survey Using Inspections as an Example. Proceeding of the First International Symposium on Empirical Software Engineering and Measurement (ESEM). Washington: IEEE Computer Society. 2007. p. 31-40.

KANG, K. C. et al. **Feature–Oriented Domain Analysis (FODA) Feasibility**. Carnegie Mellon University. [S.l.]. 1990.

KRUEGER, C. W. Introduction to the Emerging Practice of Software Product Line Development. In: _____ **Methods and Tools**. [S.l.]: [s.n.], v. 14, 2006. p. 3-15.

KRZYSZTOF, C.; HELSEN, S.; EISENECKER, U. Formalizing cardinality-based feature models and their specialization. **Software Process: Improvement and Practice**, 2005. 7-29.

LEDCZI, A. et al. **On Metamodel Composition**. IEEE. Cidade do México: Proceedings of the 2001 IEEE International Conference on Control Applications, 2001. (CCA '01). 2011. p. 756-760.

LOCHMANN, H.; BRÄUER, M. **Towards Semantic Integration of Multiple Domain-Specific Languages Using Ontological Foundations**. Proceedings of 4th ATEM on MoDELS. Nashville: [s.n.]. 2007.

LOCHMANN, H.; HESSELLUND, A. **An Integrated View on Modeling with Multiple Domain-Specific Languages**. Proceedings of the IASTED International Conference Software Engineering (SE 2009). [S.l.]: [s.n.]. 2009. p. 1-10.

LOUGHRAN, N. et al. Language Support for Managing Variability in Architectural Models. **Proceedings of the 7th International Symposium on Software Composition**, Budapest, 29-30 Março 2008. 36-51.

MENS, ; STRAETEN, R. V. D.; D'HONDT, M. Detecting and resolving model inconsistencies using transformation dependency analysis, Genova, v. 4199, p. 200-214, 2006.

OMG. **Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification**. Object Management Group (OMG). [S.l.]. 2011.

PFLEEGER, S. L. **Experimental design and analysis in software engineering:** Part 2 - How to set up and experiment. SIGSOFT Software Engineering Notes. [S.l.]: [s.n.]. 1995. p. 22-26.

PFLEEGER, S. L. **Experimental Design and Analysis in Software Engineering:** Types of Experimental Design. SIGSOFT Software Engineering Notes. [S.l.]: [s.n.]. 1995. p. 14-16.

POHL, ; BÖCKLE, G.; LINDEN , F. J. V. D. **Software Product Line Engineering:** Foundations, Principles and Techniques. New York: Springer, 2005.

SÁNCHEZ, P. et al. Engineering Languages for Specifying Product-Derivation Processes in Software Product Lines. **Proceedings of the 1st International Conference on Software Language Engineering (SLE)**, Toulouse, 29-30 Setembro 2008. 188-207.

SIMOS, M. et al. **Organization Domain Modeling (ODM) Guidebook**. [S.l.]. 1996. STARS-VC-A025/001/00.

STAHL, T.; VOLTER, M. **Model-Driven Software Development**. Heidelberg, Germany: John Wiley & Sons, Ltd., 2005.

STEINBERG, D. et al. **EMF:** Eclipse Modeling Framework. 2ª. ed. [S.l.]: Addison-Wesley Professional, 2008.

VOELTER, M.; VISSER, E. **Product Line Engineering using**. Proceeding of the 2011 15th InternationalSoftware Product Line Conference (SPLC). Washington: IEEE Computer Society. 2011. p. 70-79.

WARMER, J.; KLEPPE, A. **Building a flexible software factory using partial domain specific models**. 6th OOPSLA Workshop on Domain-Specific Modeling (DSM 2006). Portland: University of Jyvaskyla. 2006. p. 15-22.

WEISS, D. M.; LAI, C. T. R. **Software Product-Line Engineering:** A Family-Based Software Development Process. Boston: Addison-Wesley Professional, 1999.

WOHLIN, C. et al. **Experimentation in Software Engineering:** An Intoduction. Norwell: Kluwer Academic Publishers, 2000.

ZSCHALER, S. et al. Produt-driven software product line engineering. In: RASHID, A.; ROYER, J.-C.; RUMMLER, A. **Aspect-Oriented Model-Driven Software Product Lines:** The AMPLE Way. New York: Cambridge University Press, 2011. Cap. 10, p. 287-315.

APÊNDICE A – Requisitos de domínio de experimentos controlados

Tabela 9. Requisitos do domínio de experimentos controlados

#	Título	Descrição
R01	Identificação do experimento	Um experimento deve possuir sempre um nome que o identifique
R02	Definição de um plano experimental	Um experimento deve possuir um plano experimental que definirá seus fatores, níveis e tipo de design
R03	Configuração do tipo de design	Experimentos podem possuir designs diferenciados que implicará no seu plano experimental; completamente aleatorizado, completamente aleatorizado em blocos e quadrado latino, são alguns dos designs mais comuns
R04	Definição dos fatores desejáveis	Todo experimento deve possuir ao menos um fator desejável; fatores representam variáveis de controle que podem influenciar na aplicação de um tratamento e, desta forma, indiretamente o resultado do experimento; fatores desejáveis são aqueles de interesse direto de observação em um experimento
R05	Definição de fatores indesejáveis	É possível que outros fatores secundários coexistam em um experimento e influenciem no seu resultado, mesmo que não sejam desejáveis; são fatores que precisam ser considerados nos resultados mas que não são o foco da observação experimental
R06	Definição de níveis	Um fator pode ser divido em níveis; um nível representa instâncias ou derivações de fator
R07	Definição dos processos de um experimento	Deve-se haver ao menos um processo definido para cada experimento; um processo é formado pelo conjunto de ações, papéis e artefatos que serão fornecidos e produzidos durante a execução de um experimento
R08	Identificação de um processo	Um processo é identificado por um nome e deve possuir uma série de elementos que o definam: fases, tarefas, atividades, grupos, disciplinas, etc.
R09	Definição do ciclo de vida de um processo	Todo processo precisa ter definido o seu ciclo de vida; é ele que indicará quais tarefas serão executadas, agrupadas em atividades, e em que ordem
R10	Definição de disciplinas	Opcionalmente um processo classificar seus artefatos e tarefas em agrupamentos lógicos chamados de disciplinas
R11	Definição dos papéis	Um papel em um processo é definido por um nome e uma descrição que represente uma função a ser exercida durante o processo; um processo pode conter um ou mais papéis
R12	Definição de artefatos	Um processo precisa definir seus artefatos; um artefato possuir um nome e uma descrição que o identifique e poderá ser usado como entrada ou saída de atividades e tarefas
R13	Definição de tarefas	As tarefas são os elementos base de um processo, elas descrevem ações a serem executadas por um determinado papel primário e outros adicionais; uma tarefa é identificada por um nome e pode ser quebrada em passos encadeados que auxiliem a sua execução; elas podem ainda receber artefatos de entrada e fornecer novos artefatos de saída
R14	Definição de atividades	Uma atividade possui as mesmas características e elementos de uma tarefa com a diferença que possuem uma ordem de execução e fazem parte do clico de vida do processo; uma atividade pode conter uma relação de tarefas
R15	Definição das métricas em um experimento	É preciso definir métricas em um experimento; cada métrica estará associada a um processo relacionado do experimento

R16	Identificação de uma métrica	Uma métrica é identificada por um nome e uma descrição, e precisa está relacionada a um processo
R17	Configuração de uma métrica	É preciso definir a forma de coleta de dados de uma métrica, que pode ser contínua, quando medida de forma ininterrupta, ou intercalada, que permite que a coleta da métrica seja feita em intervalos que serão somados no final para o cálculo do valor total da medida; além disso, a unidade de medida poderá ser minutos, números ou até mesmo unidades de casos de usos em um sistema
R18	Definição do objeto a ser medido	A medição de uma métrica está relacionada a algum elemento do processo na qual a mesma está associada; é possível que a medicação esteja associada a um conjunto de tarefas ou artefatos ou mesmo um intervalo de atividades
R19	Definição de questionários	Em alguns experimentos é possível que tenhamos a especificação de questionários com objetivo de coletar feedback dos participantes do experimento;
R20	Identificação de um questionário	Um questionário é identificado por um nome e seu conjunto de questões
R21	Configuração do tipo de questionário	A aplicação de um questionário pode ocorrer no início da execução do experimento ou após a sua finalização, ou, quando associado a um processo específico do experimento, antes ou depois da execução do processo relacionado
R22	Definição de questões em um questionário	As questões devem possuir um nome e uma descrição que as identifiquem
R23	Configuração do tipo de questão	Um questionário pode possuir inúmeros tipos de questões, tais como de múltipla escolha, de campo aberto, com opções em listas, etc.
R24	Definição de alternativas	As questões que possuírem alternativas deveram informa-la junto com sua definição e informar também a quantidade mínima de alternativas requeridas

APÊNDICE B – Meta-modelos das DSLs projetadas durantes o estudo de caso

Esta seção apresenta os meta-modelos das quatros DSLs projetadas durantes o estudo de caso realizado para avaliar a aplicação do método proposto neste trabalho.

B.I. Meta-modelo *ProcessDsl*

Os elementos do meta-modelo da *ProcessDsl* são apresentados na Figura 31 e em seguida é descrito algumas características dos seus principais elementos.

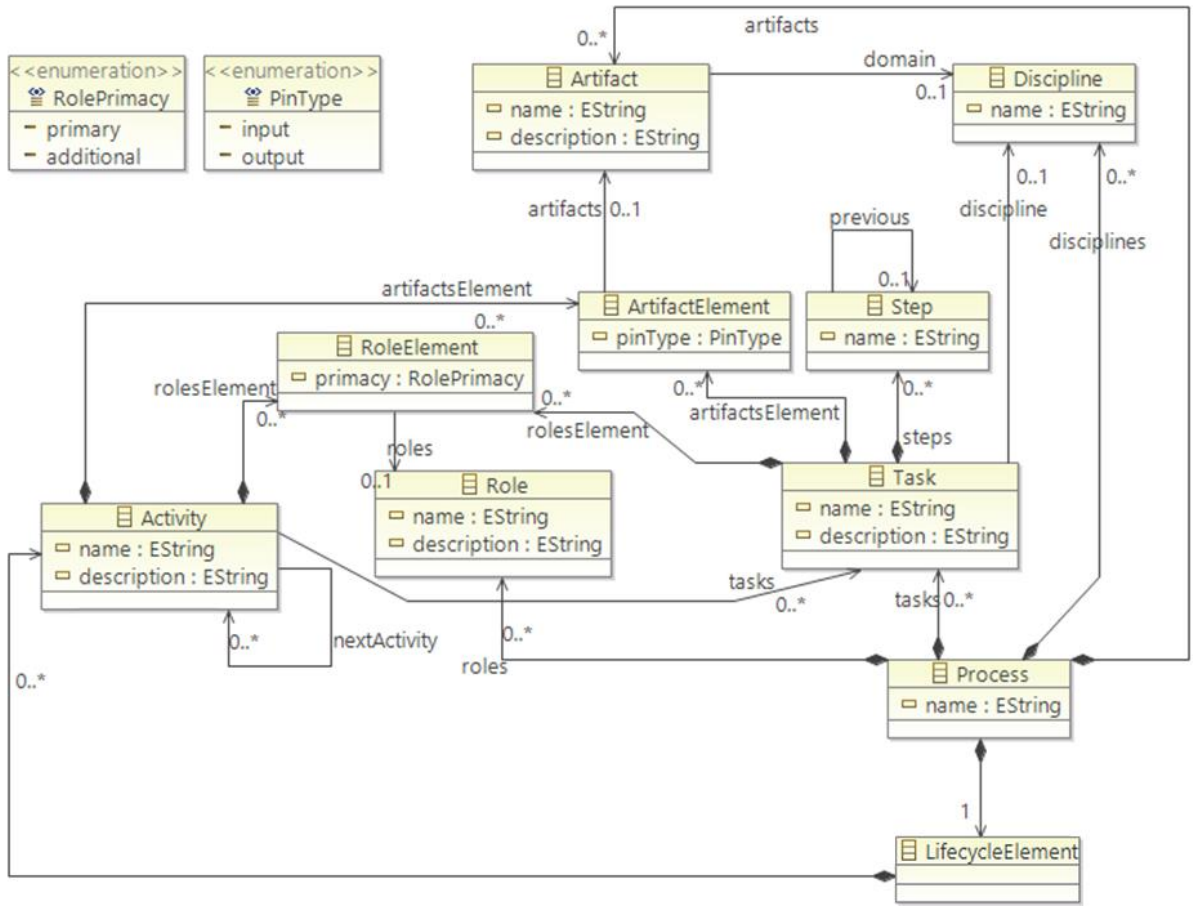


Figura 31. Meta-modelo da linguagem de processos

A seguir a descrição dos elementos do meta-modelo:

- *Process*: Elemento raiz do meta-modelo arquitetural; armazena as informações básicas do processo, tal como nome (*name*); possui uma lista *roles* do tipo *Role*, uma lista *tasks* de *Task*, uma lista *artifacts* de *Artifact*, uma

lista *disciplines* de *Discipline*; Além disso, possui um elemento do tipo *LifeCycleElement*.

- *LifeCycleElement*: Define os elementos que formam o ciclo de vida do processo; os elementos podem ser um conjunto de atividades (*activity*) ordenadas com suas respectivas tarefas ou apenas um conjunto de tarefas encadeadas.

- *Artifact*: Define os tipos de artefatos de um processo, como, por exemplo, um “Documento de Visão”, um “Diagrama de Caso de Uso”, etc.

- *ArtifactElement*: Estabelece sempre uma relação do tipo “é um” com algum artefato (*Artifact*) do processo; Possui uma enumeração (*enum*) do tipo *PinType* usada para informar se a ocorrência do artefato referenciado em uma determinada tarefa ou atividade é de entrada ou saída; Assim, pode estar contido na lista *artifactsElement* em uma *Activity* ou uma *Task*.

- *Role*: Define os papéis que atuaram no processo sem associá-lo ainda a nenhuma tarefa ou atividade específica.

- *RoleElement*: Associa uma tarefa (*Task*) ou atividade (Activity) do clico de vida do processo a um papel (*Role*) atuante no processo, definindo o tipo de participação desse naquele; O tipo de atuação é definido por uma enumeração (*enum*) do tipo *RolePrimacy* que alternar valores entre primária (*primary*) ou secundária (*additional*).

- *Discipline*: Classifica um conjunto de tarefas ou artefatos em um grupo conceitual distinto; Possui função organizacional no processo; São exemplos de disciplinas em um processo de desenvolvimento de software: “Implementação”, “Requisitos”, “Testes”, etc.

- *Task:* Representa uma tarefa a ser executada durante um processo; É identificada por um nome e uma descrição e pode ser quebrada em passos (*Step*) menores para melhor orientar a sua execução; Possui uma lista *rolesElement* de *RoleElement* e uma outra lista *artifactsElement* de *ArtiactElement*.

- *Activity:* Uma atividade representa o último nível de um clico de vida em um processo; Pode estar contida dentro de uma iteração e uma fase ou simplesmente definida na raiz do clico; As atividades são apresentas em ordem de execução e, diferente das tarefas, precisam informar a próxima a ser executada; Uma atividade pode conter sua própria relação de *RoleElement* e *ArtifactElement* ou ser quebrada em tarefas especificadas na lista *tasks*.

B.II Meta-modelo *MetricDsl*

O segundo meta-modelo está representado na Figura 32 e é constituído dos elementos arquiteturais projetados para representar as métricas de um experimento.

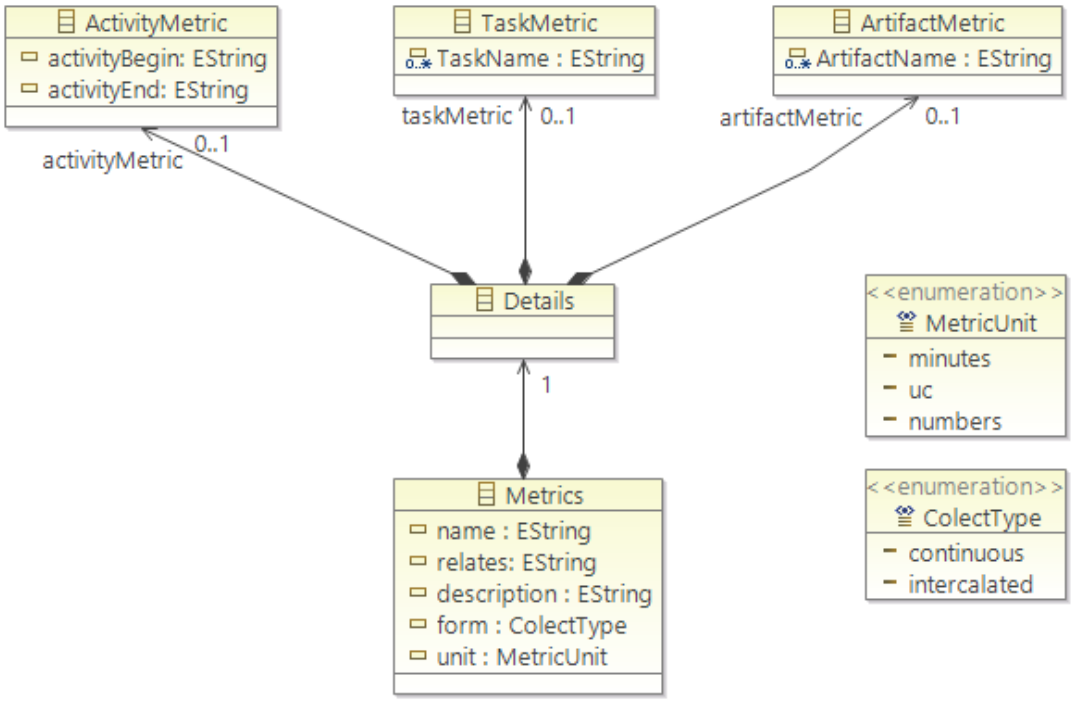


Figura 32. Meta-modelo da linguagem de Métricas

A seguir a descrição dos principais elementos de meta-modelo de métricas:

- *Metrics*: Elemento raiz do meta-modelo utiliza os atributos nome (*name*) e descrição (*description*) para identificar a métrica; possui ainda os atributos *form,* que indica como os dados serão coletados (contínuo ou intercalado), e *unit*, para configurar a unidade de medida utilizada na mensuração (minutos, casos de usos ou números); um métrica precisa está relacionada um

processo pelo atributo *relates* para poder especificar quais elementos do processo irá ser medido no elemento *details* da métrica.

- *Details*: Define os detalhes da medição da métrica, associando-a a uma *ActivityMetric* (quando a medição da métrica estiver relacionada a um conjunto de atividades), *ArtifactMetric* (quando estiver relacionada a um conjunto de artefatos) ou T*askMetric* (quando estiver relacionada a um conjunto de tarefas).

B.III Meta-modelo *QuestionnaireDsl*

A Figura 33 apresenta o meta-modelo criado para representar a arquitetura do domínio de Questionários.

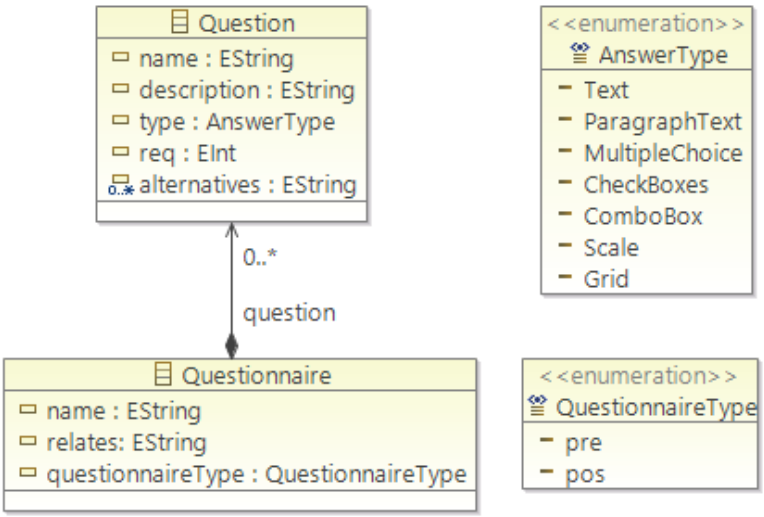


Figura 33. Meta-modelo da linguagem de Questionários

O meta-modelo é bastante simples, com apenas dois elementos:

- *Questionnaire*: Elemento raiz da meta-modelagem; possui um atributo *name* para identificar o questionário, *relates* para armazenar o nome do processo no qual o questionário está relacionado (caso possua) e *questionnaireType* para indicar se o questionário é do tipo *pre* ou *pos*.

- *Question*: Define uma questão do questionário; identificada pelos atributos *name* e *description*; o atributo *type* guarda o tipo de resposta da questão, que poderá ser: *Text*, *ParagraphText*, *MultipleChoice*, *CheckBoxes*, *ComboBox*, *Scale* ou *Grid*; o atributo *req* armazena a quantidade mínima de alternativas

que precisam ser marcadas ou respondidas (dependerá o tipo de resposta escolhido); e o atributo *alternatives* que poderá conter a relação de alternativas.

B.IV. Meta-modelo *ExperimentDsl*

Por fim, o meta-modelo do domínio de Experimentos é apresentado pela Figura 34 a seguir.

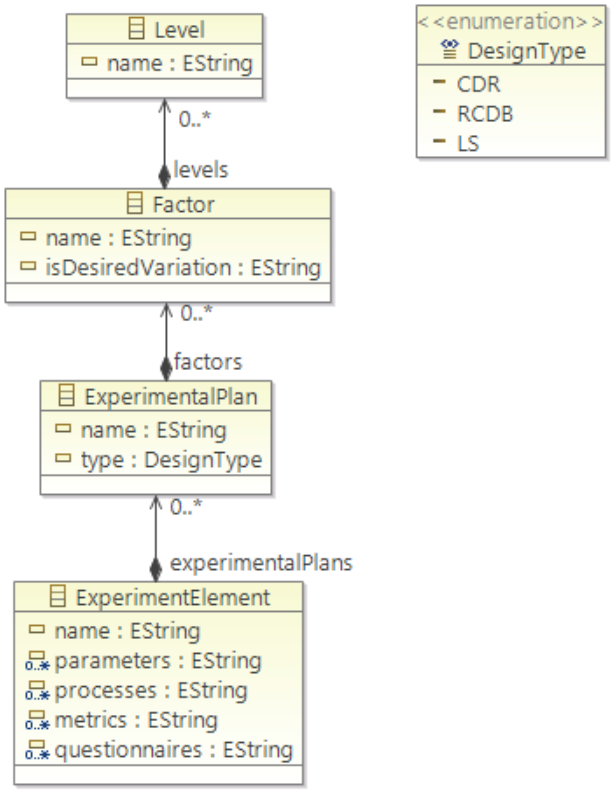


Figura 34. Meta-modelo da linguagem de Experimentos

A seguir são apresentados mais detalhes sobre os elementos desta linguagem:

- *ExperimentElement*: Elemento central da modelagem de um experimento; possui um atributo *name* que o identifica, uma lista *parameters* para armazenar informações adicionais úteis para a execução, como exemplo observações e recomendações, e mais três listas (*processes*, *metrics* e *questionnaires*) que armazenam os nomes, respectivamente, dos processos, métricas e questionários relacionados ao experimento modelado; por fim, possui uma lista com os designs dos planos experimentais (*ExperimentPlan*).

- *ExperimentPlan*: Recebe um nome (*name*) para identificar o plano e configura o tipo de experimento com o atributo *type*, que pode ser representar um design Completamente Aleatorizado (*Completely Randomized Design* – CRD), Completamente Aleatorizado em Blocos (*Randomized Complete Block Design* – RCBD) e Quadrado Latino (*Latin Square* – LS); além disso, o plano experimental possui uma lista com os fatores do experimento (*Factor*).

- *Factor*: Possui um nome (*name*) e a indicação se o fator é desejável ou não para o experimento, configurada no atributo *isDesiredVarition*; por exemplo, em um experimento cujo objetivo é comparar o desempenho das duas ferramentas para resolução de uns problemas específicos, tanto das ferramentas como os problemas serão fatores desse experimento, sendo o primeiro deles o fator desejável, por ser aquele que queremos investigar o desempenho. Um fator pode ainda ser divido em níveis (*Level*).

- *Level*: Define o nome do nível de um determinado fator; considerando o exemplo do fator ferramenta, a ferramenta A e a ferramenta B representam os níveis desse fator.