Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Hephaestus-PL: uma Linha de Produtos de Ferramentas para Linha de Produtos de Software

Lucinéia Turnes

Brasília
2012

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Hephaestus-PL: uma Linha de Produtos de Ferramentas para Linha de Produtos de Software

Lucinéia Turnes

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientador
Prof. Dr. Vander Alves

Coorientador
Prof. Dr. Rodrigo Bonifácio

Brasília
2012

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenador: Prof.ª Dr.ª Mylene Christine Queiroz de Farias

Banca examinadora composta por:

Prof. Dr. Vander Alves (Orientador) — CIC/UnB
Prof. Dr. Paulo Henrique Monteiro Borba — Centro de Informática/UFPE
Prof.ª Dr.ª Genaína Nunes Rodrigues — CIC/UnB

CIP — Catalogação Internacional na Publicação

Turnes, Lucinéia.

Hephaestus-PL: uma Linha de Produtos de Ferramentas para Linha de Produtos de Software / Lucinéia Turnes. Brasília : UnB, 2012.
99 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2012.

1. Ferramentas de Linha de Produtos de Software, 2. Gerenciamento de Variabilidades em Vários Artefatos, 3. Bootstrapping, 4. Metaprogramação, 5. Ferramenta Hephaestus

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Hephaestus-PL: uma Linha de Produtos de Ferramentas para Linha de Produtos de Software

Lucinéia Turnes

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof. Dr. Vander Alves (Orientador)
CIC/UnB

Prof. Dr. Paulo Henrique Monteiro Borba
Centro de Informática/UFPE

Prof.ª Dr.ª Genaína Nunes Rodrigues
CIC/UnB

Prof.ª Dr.ª Mylene Christine Queiroz de Farias
Coordenador do Mestrado em Informática

Brasília, 21 de junho de 2012

Agradecimentos

Este trabalho é o resultado de um esforço e aprendizagem proporcionados a mim e vindos de várias pessoas que, de forma direta e indireta, contribuíram para a sua realização.

Agradeço a Deus pela vida. Àos meus pais pelo amor e dedicação com que me ensinaram os valores da vida e me proporcionaram uma educação que foi a base da minha formação pessoal e profissional. Às minhas irmãs Luciana e Joice que sempre me incentivaram a buscar novos desafios. Aos meus sobrinhos Bárbara, Matheus, Manoela e Beatriz e meus cunhados Zulmar e Jean agradeço o carinho e a torcida por mim.

Ao meu marido João Carlos, uma pessoa muito especial que sempre teve a maior paciência e compreensão comigo e sempre esteve ao meu lado me apoiando e me ajudando no que fosse preciso. Obrigada meu bem!

Ao meu orientador, professor Vander Alves, pela orientação de qualidade que recebi, por sua disponibilidade, motivação e amizade, pela condução da pesquisa e pelos momentos de discussão e reflexão proporcionados e que contribuíram para o enriquecimento do meu aprendizado durante a realização deste trabalho. Estendo este mesmo agradecimento ao meu coorientador, professor Rodrigo Bonifácio, que sempre participou ativamente do meu trabalho, das discussões e reuniões, sempre com excelentes idéias e sugestões que contribuíram para o enriquecimento e resultados alcançados nesta pesquisa.

Ao professor Ralf Lämmel da Universität Koblenz-Landau, que teve uma participação muito relevante na concepção deste trabalho contribuindo na implementação inicial que foi refinada por mim posteriormente, gostaria de registrar meu agradecimento todo especial.

Aos colegas que conheci durante o mestrado da UnB, em especial aos colegas de orientação Idarlan, Giselle, Vinícius, Paula e Anderson agradeço a amizade, o companheirismo, a convivência e o aprendizado compartilhado.

Aos professores do curso de Mestrado em Informática do Departamento de Ciência da Computação da UnB, agradeço pelos ensinamentos e formação de qualidade que recebi nas disciplinas do curso de pós-graduação. E, aos servidores da Secretaria do Mestrado em Informática agradeço a atenção, presteza e amizade.

Finalmente, agradeço aos colegas da Secretaria de Logística e Tecnologia da Informação (SLTI) do Ministério do Planejamento, em especial ao meu diretor Corinto Meffe e ao meu coordenador Everson Lopes de Aguiar, pela oportunidade, confiança, tolerância e flexibilidade que proporcionaram viabilidade para a realização deste trabalho. Gostaria de agradecer também Cláudio Cavalcanti, que foi meu primeiro coordenador quando cheguei na SLTI em 2010, e sempre me deu total apoio para continuar no mestrado apesar das dificuldades de conciliar a pesquisa e as disciplinas com o trabalho do dia-a-dia.

Abstract

Tool support is essential for application engineering in software product lines. Despite a myriad of existing tools, most still lack adequate support for configurability and flexibility, so that it is hard for them to be applied in different contexts, e.g., addressing variability in an arbitrary combination of different artifacts and introducing and managing variability in new artifacts. Addressing this issue requires systematically exploring underlying commonality and adequately managing variability of such tools.

Accordingly, we have conducted a comparative analysis of variability management techniques for SPL tool development in the context of the SPL Hephaestus tool. The analysis reveals that two techniques, one annotative and another transformational, are most suitable to variability management in Hephaestus, and that their combination is a feasible strategy to improve such management.

Furthermore, we present domain analysis, design, implementation, and a supporting process for extending Hephaestus-PL, a software product line of software product line tools whose variability management was implemented by transformational approach using metaprogramming operations. Hephaestus-PL is supported by a process allowing instantiating product line tools for modeling variability in new and in any combination of artifacts, and has been developed by bootstrapping previous versions of the Hephaestus tool. This process supports the reactive approach and flexibility to add new assets increasing the configurability of Hephaestus-PL and reaching the goal of enabling the generation of different instances of Hephaestus-PL.

An assessment of the proposed solution reveals that it has improved configurability and flexibility when compared to previous evolution of Hephaestus.

Keywords: Software Product Line Tools, Multi-artifact variability management, Bootstrapping, Metaprogramming, Hephaestus tool

Resumo

Suporte ferramental é essencial para a Engenharia de Aplicação em Linhas de Produto de Software (LPS). Apesar de uma variedade de ferramentas existentes, a maioria delas não apresenta suporte adequado à configurabilidade e flexibilidade. Assim sendo, é difícil para elas serem aplicadas em diferentes contextos, por exemplo, endereçar variabilidade em diferentes combinações de artefatos e permitir a inserção e o gerenciamento de variabilidades de novos artefatos de diferentes domínios. Para abordar esta questão, é necessário explorar sistematicamente a comunalidade e, adequadamente, gerenciar a variabilidade de tais ferramentas.

Nesse sentido, realizamos uma análise comparativa de técnicas de gerenciamento de variabilidades para o desenvolvimento de ferramentas de LPS no contexto da ferramenta Hephaestus. A análise revela que duas técnicas, uma anotativa e outra transformacional, são as mais adequadas ao gerenciamento de variabilidades em Hephaestus, e que a sua combinação é uma estratégia viável para melhorar esse gerenciamento.

Além disso, apresentamos a análise, projeto e implementação do domínio e um processo que suporta a evolução de Hephaestus-PL, uma linha de produtos de ferramentas de linha de produtos de software onde o gerenciamento de variabilidades foi implementado por abordagem transformacional usando operações de metaprogramação. Hephaestus-PL suporta um processo que permite a instanciação de ferramentas de linha de produtos modelando a variabilidade em novos e em qualquer combinação de artefatos, e foi desenvolvida por *bootstrapping* de versões da ferramenta Hephaestus. Este processo suporta a abordagem reativa e a flexibilidade para introduzir novos ativos aumentando a configurabilidade de Hephaestus-PL e permitindo a geração de diferentes instâncias de Hephaestus-PL.

Uma avaliação da solução proposta revela que a mesma melhorou a configurabilidade e flexibilidade quando comparamos com as evoluções anteriores de Hephaestus.

Palavras-chave: Ferramentas de Linha de Produtos de Software, Gerenciamento de Variabilidades em Vários Artefatos, Bootstrapping, Metaprogramação, Ferramenta Hephaestus

Sumário

Lista de Figuras

Capítulo 1

Introdução

Linha de Produtos de Software (LPS) é um paradigma da Engenharia de Software que propõe o desenvolvimento de sistemas voltado ao reuso estratégico (Clements e Northrop, 2001; Klaus Pohl e van der Linden, 2005). Surgiu para atender as necessidades de mercado no sentido de oferecer diversidade e customização de produtos em um domínio de negócio a longo prazo, com maior agilidade e eficiência do que o desenvolvimento de produtos individuais. Nesse sentido, uma Linha de Produtos de Software (LPS) é uma família de sistemas (Parnas, 1976) gerados sistematicamente a partir de componentes reutilizáveis que representam funcionalidades comuns e variáveis voltadas a um segmento de mercado específico (Clements e Northrop, 2001).

Inicialmente, o desenvolvimento de uma Linha de Produtos de Software é um processo mais demorado do que o desenvolvimento de um sistema individual específico pois requer um esforço maior para a análise da comunalidade e variabilidade do domínio de negócio. No entanto, a longo prazo, a LPS permite a geração de novos sistemas (produtos) de forma mais otimizada do que o desenvolvimento de um sistema individual (Klaus Pohl e van der Linden, 2005). Dessa forma, observa-se como benefícios potenciais da LPS a melhoria da produtividade com menores custos de desenvolvimento e tempo de lançamento dos produtos no mercado (*time-to-market*); bem como o aumento da qualidade proporcionando maior satisfação aos clientes que poderão adquirir produtos que atendam especificamente às suas necessidades a um menor custo.

Os principais desafios do desenvolvimento de LPS são o gerenciamento da variabilidade e as estratégias de adoção (Krueger, 2001). O gerenciamento da variabilidade define como os artefatos comuns e variáveis serão representados e manipulados para a geração de instâncias de produtos a partir de uma configuração de features selecionadas da linha de produtos. As estratégias de adoção representam a abordagem utilizada para o desenvolvimento da LPS e são classificadas em proativa, extrativa e reativa (Krueger, 2001).

O suporte ferramental em Linha de Produtos de Software é essencial para apoiar as atividades do processo da Engenharia de Aplicação. A Engenharia de Aplicação é o processo de geração de instâncias de produtos da LPS segundo as necessidades e requisitos fornecidos pelo usuário a partir de uma combinação válida de features (*Configuração de Produto*). O processo de derivação de produtos utiliza o modelo de features e o conhecimento da configuração para gerar instâncias de produtos. Dado a inerente complexidade e necessária coordenação no processo de derivação (Griss, 2000), esta atividade é lenta

e propensa a erros. Para apoiar as atividades da Engenharia de Aplicação e alcançar os benefícios potenciais da LPS, ferramentas são normalmente desenvolvidas com o objetivo de sistematizar e automatizar o processo de derivação de produtos de LPS dando maior agilidade e produtividade ao processo.

1.1 Problema

Normalmente, as ferramentas de derivação de produtos são construídas para atender à diferentes tipos de artefatos de linha de produtos. A cada nova necessidade de atender à outros artefatos de linha de produtos, é demandada a geração de uma nova versão da mesma ferramenta, o que requer a sua extensão para suportar o novo artefato de linha de produtos. A geração dessa nova configuração para a ferramenta de derivação de produtos é uma tarefa manual, dependente de especialistas da ferramenta, tediosa e propensa a erros. Ou seja, pode introduzir erros na ferramenta e nos produtos dela gerados, inclusive nos produtos gerados de LPS já existente na ferramenta. Esses defeitos podem levar a resultados indesejáveis como mudança no comportamento da ferramenta, falta de reuso sistemático e consequente incorretude nos produtos gerados. Tudo isso impacta nos usuários da LPS e compromete a confiabilidade e a corretude do processo de derivação de produtos da LPS.

Nesse contexto, e em função dos benefícios da utilização de suporte ferramental para aumentar a produtividade do desenvolvimento de software, observamos o crescente aumento na utilização dessas ferramentas para atender necessidades específicas de diferentes usuários e organizações. Para isso, é importante que essas ferramentas sejam facilmente customizáveis, flexíveis e adaptáveis para suportar diferentes configurações e domínios de negócio.

A maioria das ferramentas existentes para atender à Engenharia de Aplicação em Linhas de Produto de Software não apresenta um suporte adequado aos requisitos de configurabilidade e flexibilidade (Rabiser et al., 2010). A intensidade de pesquisa nesse sentido demonstra a relevância do tema e a busca por mecanismos para suportar esses requisitos em ferramentas de derivação de produtos (Rabiser et al., 2010). Configurabilidade é a propriedade que permite à ferramenta lidar com qualquer combinação de diferentes artefatos e assim ser aplicada em diferentes contextos de domínio de negócio (Czarnecki e Eisenecker, 2000; Rabiser et al., 2010). Por exemplo, a geração de instâncias da ferramenta para cada combinação específica de artefatos casos de uso, requisitos, código-fonte e processos de negócio. A flexibilidade é outra propriedade importante para atender às mudanças das necessidades dos usuários e a evolução contínua da linha de produtos visando incorporar novos domínios ou estender linhas de produtos existentes em ferramentas de derivação de produtos. Por exemplo, inserir o tratamento de novos artefatos tais como modelos arquiteturais. Adicionalmente, a modularização de features é um requisito desejável pois facilita o gerenciamento da variabilidade e a evolução de ferramentas de derivação de produtos.

1.2 Solução Proposta

Para entender melhor o problema descrito na Seção 1.1 este trabalho analisou diferentes versões de uma ferramenta de derivação de produtos, chamada *Hephaestus* (Bonifácio, 2010). Foram estudadas três versões da ferramenta que correspondem ao seu desenvolvimento inicial e mais duas evoluções da ferramenta para atender diferentes domínios de negócio (*ativos*). A versão inicial da ferramenta foi desenvolvida para atender a derivação de produtos em especificações baseadas em cenários de casos de uso. Posteriormente, a ferramenta evoluiu para atender requisitos e código fonte. A terceira versão analisada de Hephaestus suporta linha de produtos de processos de negócio (Machado et al., 2011).

Com os objetivos de melhorar o suporte à configurabilidade e flexibilidade de ferramentas de derivação de produtos, eliminar os riscos da geração manual de variantes de ferramentas de LPS e conseguir os benefícios potenciais do desenvolvmento de LPS aplicados às ferramentas, propomos tratar a própria ferramenta como uma linha de produtos (Grünbacher et al., 2008). Assim sendo, fez-se necessária a definição de um tratamento sistemático da sua variabilidade, ou seja, explorar a comunalidade e, adequadamente, gerenciar a variabilidade de tais ferramentas para suportar a geração de instâncias da ferramenta para diferentes combinações de LPS.

Atualmente, existem outras versões de Hephaestus para atender domínios como Simulink (Steiner et al., 2012) entre outros. Entretanto, consideramos que as três versões apresentadas são suficientes para o levantamento da comunalidade e variabilidade existentes na ferramenta de derivação de produtos Hephaestus e, consequentemente, para a definição de um mecanismo que permita gerenciar a variabilidade em ferramentas de LPS oferecendo suporte à configurabilidade e flexibilidade.

Usando uma abordagem extrativa (Krueger, 2001) foi realizada a análise do domínio das versões de Hephaestus. Em seguida, foram mapeadas as features comuns e variáveis e os tipos de variabilidade encontrados e suas características como nível de granularidade e localidade. Identificamos que para o gerenciamento da variabilidade encontrada nas três versões de Hephaestus é necessário utilizar uma abordagem que suporte a extensibilidade tanto de tipos de dados quanto de funções (Lämmel e Ostermann, 2006). Nesse sentido, definimos que Hephaestus requer uma técnica de gerencimento de variabilidade que suporte tipos de dados e funções *abertas*. Entende-se por tipos de dados e funções *abertas* quando essas estruturas podem ser facilmente estendidas para atender à seleção de uma feature variável (Lämmel e Ostermann, 2006; Wadler, 1998; Visser, 1997). Nesse contexto, foi realizado um estudo comparativo com diferentes abordagens e técnicas de gerenciamento de variabilidade em LPS para avaliar sua aplicabilidade no tratamento das variabilidades em ferramentas de derivação de produtos, tais como, Hephaestus, mas passível de extensão a outras ferramentas de LPS com abstrações e variabilidades similares a Hephaestus. Usamos o método *Goal Question Metric (GQM)* (Basili et al., 1994) de avaliação com métricas qualitativas que permitiram avaliar atributos de alto nível importantes ao projeto da linha de produtos Hephaestus mesmo antes da sua implementação.

A partir dos resultados obtidos identificamos que as abordagens anotativa e transformacional seriam as mais expressivas para o desenvolvimento da linha de produtos Hephaestus (Hephaestus-PL). No entanto, devido aos problemas da falta de legibilidade e modularidade da abordagem anotativa decidimos por não utilizar esta abordagem e sim utilizar a abordagem transformacional aplicando a técnica de metaprogramação na própria

linguagem Haskell, que é a mesma linguagem de desenvolvimento da ferramenta Hephaestus. Inicialmente foi desenvolvido um protótipo de Hephaestus-PL usando o conceito de *bootstrapping* e metaprogramação em Haskell. Em seguida, realizamos o desenvolvimento de Hephaestus-PL especificando e aprimorando a proposta inicial para atender aos diferentes ativos das versões de Hephaestus analisadas e garantir as propriedades de configurabilidade e flexibilidade na ferramenta Hephaestus.

Neste caso, nós também definimos um modelo GQM para avaliar a configurabilidade e a flexibilidade em Hephaestus e Hephaestus-PL e observamos, por exemplo, que a configurabilidade é um processo automático em Hephaestus-PL sem nenhuma intervenção manual. Isso diminui os riscos de inserir defeitos decorrentes de uma configuração manual como ocorre em Hephaestus. Da mesma forma, na integração de novos ativos inseridos nas ferramentas Hephaestus e Hephaestus-PL, observamos que Hephaestus-PL oferece um gerenciamento modular de ativos o que representa homogeneidade e localidade do código do ativo. Enquanto em Hephaestus, não observamos um gerenciamento modular de ativos pois existe heterogeneidade e espalhamento do código do ativo por vários módulos da ferramenta Hephaestus.

1.3 Contribuição

A solução proposta tem as seguintes contribuições:

- análise comparativa de diferentes mecanismos de gerenciamento de variabilidade para o desenvolvimento de Hephaestus-PL, podendo ser estendida à outras ferramentas de LPS com abstrações e variabilidades similares à Hephaestus (Capítulo 3);
- análise, projeto e implementação do domínio de Hephaestus-PL, uma linha de produtos de software flexível e configurável de ferramentas de linha de produtos de software (Capítulo 4);
- definição de um processo reativo para a inserção de novos ativos e evolução de Hephaestus-PL (Capítulo 4);
- definição de um modelo de avaliação com métricas que abordam requisitos relevantes ao desenvolvimento de Hephaestus-PL, tais como configurabilidade e flexibilidade (Capítulo 4);

1.4 Estrutura

O restante desta dissertação está organizada da seguinte forma:

- Capítulo 2 apresenta a fundamentação teórica com os principais conceitos para o entendimento deste trabalho. Definimos o que é uma Linha de Produtos de Software (LPS), os principais desafios do desenvolvimento de LPS e as ferramentas de derivação de produtos, destacando a ferramenta Hephaestus abordada neste trabalho.

- Capítulo 3 corresponde a um estudo comparativo da aplicabilidade de diferentes abordagens e técnicas de gerenciamento de variabilidade em LPS para o tratamento das variabilidades de diferentes versões de Hephaestus na perspectiva da engenharia de domínio, visando promover flexibilidade e adaptabilidade à ferramenta Hephaestus (Turnes et al., 2011).
- Capítulo 4 corresponde ao detalhamento do desenvolvimento de Hephaestus-PL descrevendo a análise de domínio, projeto e implementação das features e do gerenciamento de variabilidades usando uma abordagem transformacional com a técnica de metaprogramação em Haskell. Além disso, apresenta a definição de um processo reativo para a inserção de novos ativos e evolução de Hephaestus-PL e o resultado de um modelo de avaliação proposto com métricas que abordam os requisitos de configurabilidade e flexibilidade relevantes ao desenvolvimento de Hephaestus-PL.
- Capítulo 5 resume as contribuições, apresenta as conclusões desta pesquisa e discute trabalhos relacionados e propostas de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo apresentamos conceitos e aspectos importantes que fundamentam a teoria de Linha de Produtos de Software utilizada no nosso trabalho. Na Seção 2.1 definimos o conceito de Linha de Produtos de Software e apresentamos seus principais artefatos representados pelo Modelo de Features (Seção 2.1.1), *Asset Base* (Seção 2.1.2), Conhecimento de Configuração (Seção 2.1.3) e Configuração de Produto (Seção 2.1.4). A Seção 2.2 apresenta os principais desafios relacionados à abordagem Linha de Produtos de Software que são, respectivamente, o gerenciamento de variabilidades (Seção 2.2.1) e as estratégias de adoção (Seção 2.2.2). Por fim, a Seção 2.3 descreve o processo de derivação de produtos e a importância do suporte ferramental para apoiar as atividades e a automação desse processo. Ainda nesta seção são apresentados, de forma sucinta, os requisitos mais relevantes a serem observados no suporte ferramental de derivação de produtos para atender à Engenharia de Aplicação de LPS, conforme resultado do levantamento de uma Revisão Sistemática da Literatura (Rabiser et al., 2010). A ferramenta Hephaestus que foi utilizada na solução proposta deste trabalho é apresentada na Seção 2.3.1.

2.1 Linha de Produtos de Software

Uma Linha de Produtos de Software (LPS) é um conjunto de sistemas de software que compartilham um conjunto comum e gerenciado de features para satisfazer necessidades específicas de um segmento particular de mercado ou missão e que são desenvolvidos a partir de um conjunto de artefatos comuns (*core assets*) pré-definidos (Clements e Northrop, 2001).

LPS é uma abordagem que permite construir softwares customizáveis e extensíveis de forma sistemática a partir de artefatos gerenciáveis e reutilizáveis. Nesse sentido, o desenvolvimeto de Linha de Produtos de Software apresenta como benefícios a redução nos custos de desenvolvimento de um produto, consequência da reutilização de componentes, e a redução do tempo de entrega de produtos (*time-to-market*). No desenvolvimento de produtos que não utilizam LPS, esse tempo é considerado constante e corresponde ao tempo gasto no desenvolvimento do produto. Já na engenharia de linha de produtos, esse tempo é inicialmente maior devido a construção dos artefatos comuns e variáveis. Depois disso, esse tempo de entrega do produto tende a ser reduzido, uma vez que novos produtos são desenvolvidos com o reuso estratégico e sistemático de artefatos existentes, ou seja, os ciclos de desenvolvimento são mais curtos (Klaus Pohl e van der Linden, 2005).

Consequentemente, observa-se relevante aumento da produtividade usando LPS. Além disso, observa-se também como benefício da LPS a melhoria da qualidade de software gerado decorrente do reuso de artefatos que são revistos e testados em muitos produtos e isso implica que as falhas encontradas são corrigidas, aumentando a qualidade desses produtos.

É crescente a utilização de LPS na indústria em áreas como: jogos ou softwares para dispositivos portáteis como celulares visando atender a grande variedade de dispositivos celulares existentes no mercado; aplicações embarcadas de tempo real que suportam funções operacionais do piloto usadas na indústria aeronáutica; softwares para dispositivos de TV; sistemas de informação corporativos; middlewares e IDEs customizáveis. Tudo isso demonstra a relevância da abordagem LPS para atender às crescentes e diferentes demandas do mercado.

Para modelar e especificar a similaridade e a variabilidade de uma Linha de Produtos de Software e sistematizar o processo de derivação de produtos são definidos os seguintes artefatos: Modelo de Features (Seção 2.1.1), *Asset Base* (Seção 2.1.2), Conhecimento de Configuração (Seção 2.1.3) e Configuração do Produto (Seção 2.1.4).

2.1.1 Modelo de Features

Uma Linha de Produtos de Software é especificada, modelada e implementada em termos de suas features. Feature é uma propriedade ou funcionalidade que é relevante e visível para algum interessado (*stakeholder*) na LPS e que é usada para identificar similaridades ou variabilidades existentes entre os diferentes produtos/sistemas da LPS (Czarnecki e Eisenecker, 2000).

Modelos de Features (FM) são usados para modelar e especificar as comunalidades e variabilidades em famílias de sistemas e linhas de produtos definindo seu escopo (espaço do problema). O FM define o conjunto de features comuns e variáveis organizado hierarquicamente com dependências, relacionamentos e restrições entre as features. Semanticamente, modelos de features representam o conjunto de configurações correspondentes aos produtos que podem ser gerados para a linha de produtos de software.

No Modelo de Features toda feature tem uma feature *pai* exceto a feature *raiz* que fica no topo da hierarquia do Modelo de Features e simboliza a própria LPS. Semanticamente, o relacionamento de uma feature pai com suas features filhas (subfeatures) define se as subfeatures são: *(1) obrigatórias* e estão presentes em todos os produtos; *(2) opcionais* e estão presentes em alguns produtos mas não em todos; *(3) alternativas* quando exatamente uma delas está presente em cada produto e; *(4) or-feature* quando todos os produtos apresentam pelo menos uma delas podendo apresentar mais de uma subfeature deste tipo (Czarnecki e Eisenecker, 2000). A Figura 2.1 ilustra o Modelo de Features da linha de produtos *eShop* (Antkiewicz e Czarnecki, 2004), onde as features *ShippingMethod* e *SearchOptions* são *obrigatórias*; as features *ShoppingCart*, *Bonus* e *UpdateUserPreferences* são *opcionais*; *Economical*, *Fast* e *ForeignShip* são *or-features* e, por fim, *Hints* e *SimilarResults* são features *alternativas*.

As *restrições* entre features expressam critérios a serem respeitados para tornar válida uma configuração de produto que representa uma seleção de features do Modelo de Features para a geração de um produto. As restrições mais comuns são: *feature A requer feature B*, ou seja, a seleção da feature A em um produto implica a seleção da feature

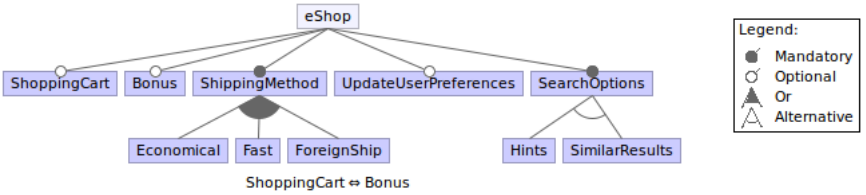

Figura 2.1: Modelo de Features *eShop*. Fonte: (Antkiewicz e Czarnecki, 2004)

B e, *feature A exclui feature B*, ou seja, nesse caso as features A e B são mutuamente excludentes e não podem aparecer no mesmo produto. A Figura 2.1 apresenta ainda a restrição $ShoppingCart \Leftrightarrow Bonus$ que significa que a feature *ShoppingCart* é selecionada se, e somente se, a feature *Bonus* for selecionada.

2.1.2 Asset Base

Asset Base é a coleção de ativos relacionados ao desenvolvimento de software para atender a uma LPS. São heterogêneos e suportam todas as fases do desenvolvimento. *Asset Base* representa a especificação e implementação de ativos como documentos de requisitos, modelos de domínio, arquitetura de software, modelos de projeto, código, classes, componentes, templates, casos de testes, arquivos de imagens, arquivos de propriedades, arquivos XML entre outros.

Estes ativos contribuem com ou representam o conjunto de artefatos reutilizáveis que implementam as features no espaço da solução da LPS e são combinados de diferentes formas para a geração de produtos.

2.1.3 Conhecimento de Configuração

O Conhecimento de Configuração (CK) é um artefato importante no processo de geração de produtos da LPS pois relaciona o espaço do problema, representado pelo Modelo de Features, com o espaço da solução da linha de produtos, representado pelo *Asset Base*, permitindo a construção sistemática de produtos. Ou seja, na prática o CK relaciona expressões de features, representadas usando lógica proposional, com os ativos da LPS que implementam as features de modo a gerar o produto final.

Existem duas formas de especificar o CK e que são definidas a partir da abordagem utilizada no desenvolvimento da LPS (Bonifácio e Borba, 2009; Neves et al., 2011). A forma mais simples e direta consiste em mapear expressões de features com os nomes dos ativos que implementam as features no espaço da solução da LPS. Essa representação do CK é aplicada em LPS que utilizam abordagem composicional de features para a derivação de produtos. Outra forma de representar o CK consiste em mapear expressões de features com transformações. As transformações irão agir sobre os ativos da instância de produto sendo gerada aplicando alguma modificação para atender à seleção de features correspondente à expressão de features em questão. Essa representação do CK é aplicada em LPS que utilizam abordagens anotativa e transformacional no processo de derivação de produtos.

O CK corresponde a uma lista de itens de configuração que representam uma tupla formada por expressão de features e ativos ou expressão de features e transformações,

conforme abordagem da LPS utilizada. O processo de derivação de produtos varre todas as linhas (itens de configuração) do CK e avalia cada expressão de features conforme as features selecionadas e representadas na configuração de produto (PC). Caso a expressão de features seja satisfeita então é executado a segunda parte da tupla, ou seja, é incorporado o código associado ao nome do ativo ou é executada a transformação correspondente à expressão de feature na instância de produto sendo gerada.

2.1.4 Configuração de Produto

Uma Configuração de Produto (PC) é qualquer subconjunto de features do Modelo de Features que representa uma seleção válida de features que satisfaz todas as restrições do Modelo de Features e onde não há variabilidade a ser tratada. A Configuração de Produto especifica em termos de features um produto da LPS a ser gerado pelo processo de derivação de produtos.

2.2 Desafios da Linha de Produtos de Software

A utilização do desenvolvimento de Linha de Produtos de Software, apesar dos benefícios alcançados a longo prazo, apresenta grandes desafios para as indústrias pois requer, inicialmente, um nível maior de esforço e riscos com investimentos altos de tempo, custos, recursos técnicos especializados e dedicados, novas ferramentas e processos.

Esses desafios representam muitas vezes uma barreira proibitiva para adotar LPS (Krueger, 2001). Nesse cenário, os interesses da organização com relação à utilização de LPS se dividem. De um lado a área de negócios que percebe um modelo estratégico lucrativo proporcionado pela LPS e, de outro lado, a área de engenharia que precisa atender com agilidade às demandas por sistema e não pode assumir riscos, alocar recursos, investir tempo e custos para o desenvolvimento de LPS.

Nesse contexto, os principais desafios do desenvolvimento de LPS são o gerenciamento da variabilidade e as estratégias de adoção (Krueger, 2001). Estes desafios estão apresentados nas Seções 2.2.1 e 2.2.2, respectivamente.

2.2.1 Gerenciamento de Variabilidades

Desenvolver Linha de Produtos de Software consiste em gerenciar adequadamente suas variabilidades. Assim sendo, o gerenciamento de variabilidades é um requisito chave no desenvolvimento de LPS para prover suporte à especificação, implementação, seleção e combinação das similaridades e variabilidades da LPS para o processo de derivação de produtos. Ou seja, promover a gerência de features comuns e variáveis de uma LPS e o reuso de uma arquitetura comum e artefatos de código para produzir diferente produtos.

O gerenciamento da variabilidade no processo de derivação de produtos define como os artefatos comuns e variáveis serão representados e manipulados para a geração de instâncias de produtos a partir de uma configuração de features selecionadas da linha de produtos.

Definir a variabilidade de uma LPS envolve modelar o espaço do problema (Modelo de Features) e o espaço da solução (*Asset Base*) e especificar o mapeamento entre o espaço do problema e o espaço da solução (Configuração do Conhecimento).

No espaço da solução as variações existem em diferentes níveis de abstração e, de acordo com a complexidade de sistemas e variedade de tecnologias usadas para implementar variabilidade, é necessário gerenciar diferentes níveis de abstração e artefatos correspondentes ao ciclo de vida do desenvolvimento do software, indo desde testes, implementação, arquitetura até requisitos. No nível de implementação, o desafio é como tratar o código das features porque normalmente as *features* tendem a ter sua implementação espalhada por vários módulos e entrelaçada com o código de outras *features* no mesmo módulo (Mezini e Ostermann, 2004; Alves et al., 2007).

Existem diferentes abordagens e técnicas associadas para o gerenciamento de variabilidades. Elas podem ser classificadas nas seguintes categorias (Kästner et al., 2008; Schaefer et al., 2010a; Voelter e Groher, 2007): composicional, anotativa, dirigida a modelos e tranformacional.

A abordagem *composicional* dá suporte à modularização física das features da LPS e a geração de produtos ocorre através da seleção e composição de módulos que implementam as features do produto desejado. Dentro dessa abordagem destacam-se duas técnicas de desenvolvimento de LPS: *Programação Orientada por Features (FOP)* (Prehofer, 1997) e *Programação Orientada por Aspectos (AOP)* (Kiczales et al., 1997). *FOP* provê suporte para que as similaridades e variabilidades sejam modularizadas e cada feature é implementada em um módulo distinto e representa um incremento na funcionalidade do sistema base (*step-wise refinement*) (Batory et al., 2003). Jak que oferece mecanismos como *mixin-layers* e colaboração (Batory, 2004), AHEAD (Díaz et al., 2005), FeatureIDE (Kästner et al., 2009) e FeatureHouse (Apel et al., 2009a) são alguns exemplos de técnicas e mecanismos baseados na composição de artefatos e que suportam o desenvolvimento de LPS usando *FOP*.

A modularização física das features apesar de proporcionar benefícios como simplificar a gerência de variabilidades e a derivação de produtos da LPS, normalmente, ela não é possível devido às características de espalhamento e entrelaçamento que as features apresentam. O *Desenvolvimento de Software Orientado por Aspectos (AOSD)* permite a modularização de interesses transversais (*crosscutting concerns*) e pode ser aplicado em vários níveis de abstrações do desenvolvimento de LPS. No nível de implementação, a *Programação Orientada por Aspectos (AOP)* provê suporte para modularização de interesses transversais e, consequentemente, pode ser usada no desenvolvimento de LPS pois permite uma melhor modularização de features que tem seu código espalhado e entrelaçado com outras features. Assim, as features se assemelham aos interesses transversais e tem seu código encapsulado e implementado em módulos separados como o conceito de *aspectos* (Alves et al., 2007). Como exemplo de linguagem *AOP* podemos citar a linguagem AspectJ (Kiczales et al., 2001).

A abordagem *anotativa* provê o uso de diretivas de pré-processamento para anotar os trechos de código associados a uma determinada feature (Kästner et al., 2008). As linguagens C e C++ já tem suporte às diretivas de pré-processamento. A geração de produtos ocorre com a definição do valor da constante simbólica das diretivas de pré-processamento associadas às features selecionadas, antes da pré-compilação, para definir a presença dos trechos de código das features selecionadas no produto gerado. *CIDE (Colored IDE)* (Kästner et al., 2009) é uma ferramenta que provê a separação virtual de features e o seu gerenciamento usando uma anotação visual que associa cores de fundo aos trechos de código das features.

Várias abordagens vem surgindo para auxiliar no processo de gerência de variabilidades e derivação automática de produtos. Muitas dessas abordagens se baseiam em técnicas de *Desenvolvimento Dirigido por Modelos (MDD)* que oferecem um suporte melhor para a gerência automatizada de variabilidades. Parte dessas ferramentas vem também investigando o uso combinado de mecanismos avançados de composição de *MDD* e *AOSD* para uma melhor modularização de variabilidades em LPS (Voelter e Groher, 2007).

Por fim, podemos citar a abordagem *transformacional* (Schaefer et al., 2010a; Xavier e Borba, 2010) que utiliza técnicas como *DSL* e *metaprogramação* para definir regras de transformações de programas para o gerenciamento de variabilidades e derivação de produtos em LPS.

2.2.2 Estratégias de Adoção

As estratégias de adoção, classificadas em *proativa*, *reativa* e *extrativa* (Krueger, 2001), definem como uma organização pode adotar o desenvolvimento de LPS.

Na abordagem *proativa*, a organização realiza a análise do domínio de negócio, o projeto e a implementação da nova LPS para suportar todo o escopo de possíveis produtos. É uma estratégia mais demorada e depende de recursos especializados (*stakeholders* e especialistas em LPS) para a identificação e modelagem do espaço do problema e implementação do espaço da solução. Essa abordagem é mais apropriada quando os requisitos do conjunto de produtos são bem definidos e estáveis. Existe um esforço maior no início do projeto mas esse esforço diminui consideravelmente uma vez que a linha de produtos fica completa. Essa abordagem não é recomendada quando houver restrições quanto ao esforço, custo e tempo necessários à técnica ou quando existir um risco alto na identificação incorreta de requisitos da linha de produtos.

Na abordagem *extrativa*, a organização extrai a LPS a partir de produtos existentes. A comunalidade e a variabilidade são extraídas do código fonte dos produtos existentes para a construção da LPS. Esta abordagem permite alto nível de reuso do código dos produtos existentes e maior agilidade no desenvolvimento da LPS comparado com a abordagem proativa.

Finalmente, a abordagem *reativa* é uma abordagem incremental que evolui a LPS para atender demandas por novos produtos ou novos requisitos de produtos existentes da LPS.

Essas abordagens não são mutuamente exclusivas pois é comum utilizar a abordagem *extrativa* ou *proativa* para construir a LPS e depois evoluí-la usando a abordagem *reativa*. Particularmente, nota-se que é mais comum a utilização da abordagem *extrativa* para o desenvolvimento da LPS e depois evoluir a linha usando a abordagem *reativa*.

2.3 Ferramentas de Derivação de Produtos

A derivação de produtos é o processo de construir um produto a partir do conjunto de artefatos de código reutilizáveis que implementam uma LPS. Esse processo envolve a seleção, composição e customização desses artefatos de código com o objetivo de atender um produto específico da LPS (configuração de features).

O uso de ferramentas de engenharia de software e processos automatizados reduzem o custo e aumentam a produtividade no desenvolvimento de sistemas. Em Linha de Produtos de Sofware, as ferramentas de derivação de produtos são desenvolvidas e utilizadas na

fase de Engenharia de Aplicação para dar suporte à automação das atividades do processo de construção de produtos por meio do reuso estratégico de uma arquitetura comum e de artefatos comuns e variáveis implementados no espaço da solução da LPS e desenvolvidos durante a Engenharia de Domínio.

As ferramentas de derivação de produtos utilizam diferentes técnicas e estratégias para especificar o espaço do problema (modelo de features), o espaço da solução (artefatos de código) e o conhecimento da configuração (mapeamento entre features e artefatos de código ou transformações sobre eles) que irão guiar o processo de geração de produtos.

Através de uma Revisão Sistemática da Literatura e avaliação de especialistas (Rabiser et al., 2010) foram identificados e validados requisitos chaves para uma ferramenta prover suporte adequado à derivação de produtos, listados abaixo segundo a ordem de sua relevância:

- *resolução interativa e automática de variabilidades* para facilitar o trabalho de seleção de features/decisões e customização de produtos pelos usuários. A variabilidade deve ser apresentada de forma simples escondendo detalhes técnicos de modelos de variabilidades complexos. Além disso, a ferramenta deve apresentar as variabilidades de forma interativa e oferecer ao usuário um *feedback* das suas escolhas e decisões com a automação das atividades do processo de derivação de produtos.

- *adaptabilidade e extensibilidade* para permitir que a ferramenta suporte diferentes domínios e organizações. As necessidades de mudança de usuários e a evolução contínua da linha de produtos motivam a flexibilidade e adaptabilidade de ferramentas de derivação do produto para atender necessidades futuras. Metamodelagem e suporte à integração de geradores de domínio específico são geralmente usados para suportar derivação de produtos em diferentes domínios, organizações e cenários tecnológicos.

- *suporte ao gerenciamento de requisitos da aplicação* para permitir uma possível evolução da linha de produtos em função de requisitos não atendidos pela linha de produtos atual.

- *visualizações de variabilidades flexíveis e específicas ao usuário*, a ferramenta de derivação de produtos não deve estar limitada a apenas uma forma de representação e visualização das variabilidades, permitindo a integração de diferentes visualizações para atender diferentes perfis de usuários.

- *guia para usuários finais* é um roteiro complementar ao requisito *resolução interativa e automática de variabilidades* que ajudará os usuários a entender melhor as escolhas a serem realizadas para a geração de produtos.

- *suporte ao gerenciamento de projetos* para permitir que a ferramenta de derivação de produtos gerencie diferentes usuários, seus papéis e responsabilidades quanto às variabilidades da linha de produtos.

É crescente a demanda pelo uso de ferramentas de derivação de produtos em diferentes organizações e domínios. Entretanto, ainda existem vários desafios e uma carência de mecanismos de implementação que atendam aos requisitos listados acima. Nesse contexto, a maioria das ferramentas de derivação de produtos não implementa todos esses requisitos de qualidade.

Para atender as necessidades específicas de usuários em diferentes domínios e considerando a ordem de relevância dos requisitos identificados na pesquisa (Rabiser et al., 2010), destacamos o requisito *adaptabilidade e extensibilidade* para suportar a customização e evolução de ferramentas de derivação de produtos para diferentes ativos. Para sermos mais objetivos, no nosso trabalho focamos no requisito *adaptabilidade e extensibilidade* interpretando-o como *configurabilidade e flexibilidade*. Assim, apresentamos uma solução que suporta a customização da ferramenta para diferentes domínios e suas combinações (*configurabilidade*) e a evolução da ferramenta de linha de produtos visando atender novos ativos e novos requisitos de ativos já disponíveis na ferramenta (*flexibilidade*).

Existem ferramentas de derivação de produtos de LPS propostas nos últimos anos como *Pure:Variants* (Beuche, 2008) baseada em modelos; *Gears* (Krueger, 2007) que implementa uma abordagem em três camadas para a derivação de produtos; *AHEAD* baseada no conceito de refinamentos sucessivos (*step-wise refinement*) para construir produtos (Batory et al., 2004); e *COVAMOF-VS* um pacote de ferramentas que implementa a abordagem *COVAMOF* (Sinnema e Deelstra, 2008), um framework de modelagem de variabilidades que suporta modelar pontos de variação e dependências em diferentes níveis, como features, arquitetura e implementação. Além disso, *COVAMOF* utiliza a notação SPEM (*Software Process Engineering Meta-model*) (OMG, 2008) para definir um processo de derivação.

Podemos citar ainda as ferramentas de desenvolvimento de LPS *Captor-AO* (Pereira et al., 2008), *CrossMDA-SPL* (Filgueira, 2009) e *GenArch* (Cirilo et al., 2012) baseadas em técnicas de desenvolvimento dirigido por modelos (MDD) e com características de desenvolvimento de software orientado por aspectos (AOSD) que permitem a gerência de variabilidades e derivação automática de produtos. Em (Torres et al., 2010a) é apresentado um estudo comparativo dessas três ferramentas de derivação dirigidas por modelos com o objetivo de investigar o uso de mecanismos de MDD e AOSD, e o benefício e potencial que tais mecanismos oferecem para lidar com gerência de variabilidades e derivação automática de produtos. É avaliado também o impacto do uso combinado das técnicas de MDD e AOSD no projeto e implementação de ferramentas de gerência de variabilidades e derivação de produtos. Foram definidos vários critérios relacionados às *(i)* características gerais de técnicas e tecnologias de MDD utilizadas na derivação/-transformação de produtos; e *(ii)* adoção de técnicas de AOSD para melhorar o suporte a gerência de variabilidades para a composição de modelos. Um dos critérios avaliados está relacionado às características de MDD das abordagens no processo de derivação de produtos e refere-se à flexibilidade e extensibilidade oferecidas pelas ferramentas para suportar a adição de novas funcionalidades visando integrar facilmente novos módulos e extensões em cada ferramenta. O resultado mostra que a ferramenta *GenArch* tem sido recentemente estendida para prover pontos de extensão para a introdução de novas Linguagens de Domínio Específico (DSL) de uma forma não invasiva, focando na capacidade de *parsing*, pós-processamento e extensão do meta-modelo. *CrossMDA-SPL* suporta flexibilidade através da instalação de novos templates e/ou novas regras de transformação e *Captor-AO* ainda não suporta flexibilidade.

Além disso, seis ferramentas de derivação de produtos *Captor* (Junior, 2006), *CIDE* (Kästner et al., 2008), *GenArch*, *Hephaestus* (Bonifácio et al., 2009), *pure::variants* e *XVCL* (Swe et al., 2002) foram analisadas no contexto de evolução de uma linha de produtos (MobileMedia (Figueiredo et al., 2008)) para avaliar usando métricas quantitativas

as propriedades de modularidade, complexidade e estabilidade nos artefatos de derivação de produtos, especificamente o CK, durante a evolução da linha de produtos (Torres et al., 2010b). A avaliação mostrou que as ferramentas *GenArch*, *pure::variants* and *Hephaestus* que utilizam um CK modularizado representado por um arquivo ou modelo separado, oferecem benefícios para a modularização e estabilidade da LPS.

2.3.1 Hephaestus

Hephaestus (Bonifácio et al., 2009) é uma ferramenta de derivação de produtos que oferece suporte ao desenvolvimento de Linha de Produtos de Software permitindo especificar e validar a sua variabilidade através do Modelo de Features. Representa um conjunto de bibliotecas e ferramentas implementadas em Haskell para o desenvolvimento de linha de produtos.

Hephaestus implementa a abordagem MSVCM (*Modeling Scenario Variability as Crosscutting Mechanisms*) (Bonifácio e Borba, 2009) que representa uma abordagem composicional e paramétrica para o gerenciamento de variabilidades em cenários de casos de uso implementada com a técnica de orientação à aspectos aplicada na linguagem funcional Haskell (Bonifácio, 2010). Atualmente, o modelo MSVCM está sendo aplicado a outros domínios como processos de negócio (Machado et al., 2011). A ferramenta Hephaestus usa o conceito de Programação Orientada à Features (FOP) apesar de existir algumas pequenas ocorrências de espalhamento e entrelaçamento de features para suportar o processo de derivação de produtos.

Hephaestus foi desenvolvido inicialmente para gerenciar variabilidades em cenários de casos de uso. Posteriormente, a ferramenta foi customizada e estendida para suportar novas necessidades de usuários em domínios específicos, no caso requisitos, código fonte e processos de negócio que representam outros artefatos do ciclo de vida do desenvolvimento de sistemas. Atualmente, Hephaestus suporta variabilidade em diferentes tipos de ativos, que vão desde processos de negócios e modelos Simulink até código fonte, e ainda tem sido usado como a ferramenta de derivação para a Linha de Produtos TaRGeT (Ferreira et al., 2010).

O processo de derivação de produtos em Hephaestus utiliza um mecanismo que combina vários artefatos de entrada da LPS para gerar o produto final. A Figura 2.2 apresenta a arquitetura lógica de Hephaestus. O módulo principal *Build* é responsável pela geração de um produto da linha de produtos representado pelo *Instance Model*, a partir de quatro artefatos de entrada: *(i) SPL Model* que representa os artefatos reutilizáveis da LPS que implementam as features comuns e variáveis do domínio de negócio; *(ii) Feature Model* que representa a variabilidade da LPS e o conjunto de produtos que podem ser gerados com os relacionamentos entre *features* e restrições globais da linha de produtos; *(iii) Product Configuration* que representa uma seleção de *features* válidas do *Feature Model* para o produto a ser gerado; e *(iv) Configuration Knowledge* que relaciona expressões de *features* às transformações que manipulam os artefatos reutilizáveis da linha de produtos.

O *Configuration Knowledge* corresponde a uma lista de itens de configuração que representam o mapeamento de expressões de *features* às transformações. O núcleo do modelo MSVCM em Hephaestus consiste na varredura do *Configuration Knowledge* avaliando-se as expressões de *features* a partir das *features* selecionadas no *Product Configuration*. Se uma expressão de *features* é satisfeita então as transformações relacionadas a ela são

aplicadas. Assim sendo, o conjunto de transformações aplicado gera automaticamente o produto da LPS. O *Instance Model* é a saída do processo *Build*, ou seja, a instância de produto gerado após a aplicação de todas as transformações do modelo conforme a definição das *features* selecionadas para o produto (*Product Configuration*).

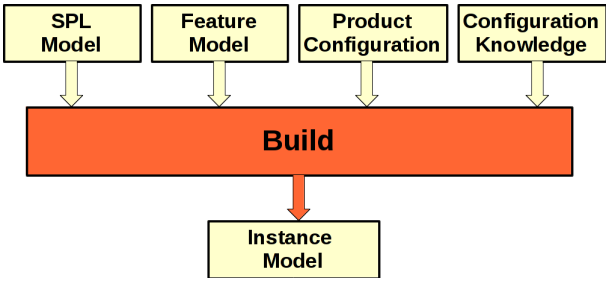

Figura 2.2: Arquitetura Lógica de Hephaestus. Fonte: (Bonifácio, 2010)

Capítulo 3

Artigo I - Técnicas para o Desenvolvimento de uma Linha de Produtos de Ferramentas de Linha de Produtos: um Estudo Comparativo

Este capítulo corresponde ao Artigo I intitulado **Técnicas para o Desenvolvimento de uma Linha de Produtos de Ferramentas de Linha de Produtos: um Estudo Comparativo** escrito por Lucinéia Turnes, Rodrigo Bonifácio e Vander Alves da Universidade de Brasília e Ralf Lämmel da Universität Koblenz-Landau (Alemanha).

Este artigo foi apresentado no *V Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS 2011)* (Turnes et al., 2011).

Resumo

Suporte ferramental é essencial para a Engenharia de Aplicação em Linhas de Produtos de Software (LPS). Apesar de uma variedade de ferramentas existentes, a maioria carece de suporte adequado à flexibilidade e adaptabilidade, de modo que é difícil para elas serem aplicadas em diferentes contextos, por exemplo, endereçar variabilidade em diferentes artefatos. Para abordar esta questão, é necessário explorar a comunalidade e, adequadamente, gerenciar a variabilidade de tais ferramentas. A fim de fornecer uma orientação sistemática nesse sentido, realizamos um estudo comparativo de técnicas de gerenciamento de variabilidade para o desenvolvimento de ferramentas LPS no contexto da ferramenta Hephaestus. A análise revela que duas técnicas, uma anotativa e outra transformacional, são as mais adequadas ao gerenciamento de variabilidade em Hephaestus, e que a sua combinação é uma estratégia viável para melhorar esse gerenciamento.

3.1 Introdução

Linha de Produtos de Software (LPS) é um conjunto de sistemas de software que compartilham um conjunto de features comuns e gerenciadas que satisfazem as necessidades

específicas de um determinado segmento de mercado ou missão e que são desenvolvidas a partir de um conjunto comum de ativos principais (*core assets*) de uma maneira prescrita (Clements e Northrop, 2001). Os benefícios potenciais alcançados em LPS incluem a melhoria da produtividade com menores custos de desenvolvimento e de tempo de lançamento no mercado (*time-to-market*) e aumento da qualidade. Para isso, o suporte ferramental para realizar as atividades é essencial, em particular, para apoiar a Engenharia de Aplicação, na qual um produto é definido pela seleção de um grupo de features e em seguida, de forma coordenada e cuidadosa , é feita a combinação de elementos de diferentes componentes envolvidos. Dado a inerente complexidade e necessária coordenação no processo de derivação (Griss, 2000), esta atividade é lenta e propensa a erros. Como resultado, a derivação de produtos individuais a partir de ativos de software compartilhados é ainda uma atividade demorada e cara em muitas organizações (Deelstra et al., 2005).

Conforme relatado por uma Revisão de Literatura Sistemática contemporânea e pesquisa de especialistas (Rabiser et al., 2010), um requisito chave para ferramentas de derivação de produtos é a flexibilidade e adaptabilidade que o mesmo estudo identifica como uma falha da maioria das ferramentas existentes. Estas devem ser adaptadas a diferentes contextos, por exemplo, tratamento com diferentes artefatos. Além disso, a mudança das necessidades dos usuários e a evolução contínua da linha de produtos motiva ainda mais a flexibilidade e adaptabilidade de ferramentas de derivação do produtos para atender às necessidades futuras. Portanto, para fornecer flexibilidade e adaptabilidade torna-se necessário gerenciar adequadamente a variabilidade dentro dessas ferramentas tratando as próprias ferramentas como LPS, como também foi sugerido por Grünbacher et al. (Grünbacher et al., 2008).

Assim, este trabalho apresenta uma análise comparativa dos mecanismos de gerenciamento de variabilidades para o desenvolvimento de LPS no contexto de ferramentas de LPS, particularmente adequados para uso em linguagens funcionais. Nós exploramos estes mecanismos no contexto das variantes existentes de Hephaestus (Bonifácio et al., 2009), uma ferramenta de LPS desenvolvida em Haskell e originalmente destinada ao gerenciamento de variabilidade em requisitos, mas que evoluiu para tratar variabilidade em diferentes tipos de artefatos. Uma descrição detalhada de Hephaestus incluindo cenários de uso pode ser encontrada em (Bonifácio et al., 2009). A análise avalia tais mecanismos e propõe uma estratégia mais adequada para o gerenciamento da variabilidade dentro do desenvolvimento de diferentes versões dessa ferramenta. Resultados revelam que duas técnicas CIDE e Stratego/XT são mais adequadas para o gerenciamento da variabilidade em Hephaestus, cuja variabilidade está descrita na Seção 3.2, e que a combinação dessas duas técnicas é uma estratégia viável.

O restante deste capítulo está organizado da seguinte forma. Seção 3.2 descreve brevemente Hephaestus e sua evolução para endereçar diferentes artefatos. Seção 3.3 apresenta a configuração do processo que orientou a análise comparativa das técnicas estudadas, que é realizada na Seção 3.4. Seção 3.5 apresenta os trabalhos relacionados e a Seção 3.6 relata as considerações finais.

3.2 Hephaestus

Hephaestus (Bonifácio et al., 2009) é uma ferramenta de derivação de produtos (Deelstra et al., 2005) disponível publicamente[1], que recebe contribuições de diferentes instituições (Universidade Federal de Pernambuco, Universidade de São Paulo, Universidade de Brasília). Inicialmente desenvolvido como uma ferramenta prova de conceito para o gerenciamento de variabilidades em cenários de casos de uso, Hephaestus fornece uma especificação declarativa (código Haskell) do estilo composicional para resolver variabilidade em LPS usando MSVCM (Modelagem de Variabilidade de Cenários como Mecanismos Transversais) (Bonifácio e Borba, 2009). Atualmente, Hephaestus suporta variabilidade em diferentes tipos de ativos, que vão desde processos de negócios e modelos Simulink até código-fonte, e Hephaestus tem sido usado como a ferramenta de derivação para a Linha de Produtos TaRGeT (Ferreira et al., 2010).

Para atender ao propósito inicial da ferramenta nós implementamos primeiro:

- tipos de dados específicos que representam modelos de caso de uso, modelos de feature e modelos de conhecimento de configuração (Czarnecki e Eisenecker, 2000), que relaciona expressões de features em lógica proposicional à transformações que tratam com a variabilidade em casos de uso.
- funções específicas que resolvem variabilidades de LPS em cenários de casos de uso, selecionando casos de uso ou cenários de um modelo LPS e resolvendo os parâmetros de acordo com configurações de features específicas. Além disso, Hephaestus fornece uma função *build* que se comporta como um interpretador para o conhecimento da configuração e é responsável pela construção de um produto específico dada uma seleção de features (ou configuração de features).

A Figura 3.1 apresenta um trecho de código da implementação inicial de Hephaestus, destacando o tipo de dados do conhecimento de configuração (linhas 3-7), o interpretador correspondente a função *build* (linhas 9-16) e a assinatura das funções de transformação (linha 1). Além disso, este trecho de código também mostra a representação inicial dos tipos de dados *SPLModel* (linhas 18-21) e *InstanceModel* (linhas 23-26), bem como a função *exportProduct* que gera uma representação em LaTeX de um produto específico de modelo de caso de uso (linhas 28-32).

3.2.1 Primeira Evolução de Hephaestus

Hephaestus foi estendido para usar MSVCM na prática. Assim, a partir de um protótipo para a experimentação com MSVCM, nós evoluímos Hephaestus para uma ferramenta que pudesse ser usada por estudantes e profissionais. Além disso, dentro de um curto período de tempo, tivemos que estender Hephaestus em outra direção de modo que pudesse gerenciar a variabilidade não somente em cenários de casos de uso, mas também em requisitos de alto nível e de código-fonte, este último através da seleção de arquivos específicos que deveriam ser compilados no início do pré-processamento para resolver variabilidades no código-fonte. Naquele tempo, Hephaestus passou a ser usado como substituto de uma

[1] http://bit.ly/iRMMZM

```
type Transformation =SPLModel →InstanceModel →InstanceModel

type ConfigurationKnowledge =[ConfigurationItem]
data ConfigurationItem = ConfigurationItem {
  expr = FeatureExpression ,
  transformations = [Transformation]
}

build fm fc ck spl = derive ts spl emptyInstance
where
    ts =concat [transformations c | c ∈ ck, eval fc (exp c)]
    emptyUCM =...
    emptyInstance =...

derive [] spl product =product
derive (t:ts) spl product = derive ts spl (t spl product)

data SPLModel =SPLModel {
   splFeatureModel :: FeatureModel,
   splUseCaseModel :: UseCaseModel
}

data InstanceModel =InstanceModel {
    featureConfiguration :: FeatureConfiguration ,
   useCaseModel :: UseCaseModel
}

exportProduct :: Path → InstanceModel → IO ()
exportProduct t product =do
  exportUcmToLatex (t ∪ "/doc.tex") (ucm product)

exportUcmToLatex =...
```

Figura 3.1: Trecho de código da implementação inicial de Hephaestus

ferramenta proprietária para gerenciar variabilidades na Linha de Produtos TaRGeT (Ferreira et al., 2010), e novos ativos de produto deveriam ser exportados como conseqüência do processo de "build" de Hephaestus. Nós mostramos algumas das features relacionadas com esta versão nas Figuras 3.2 e 3.3.

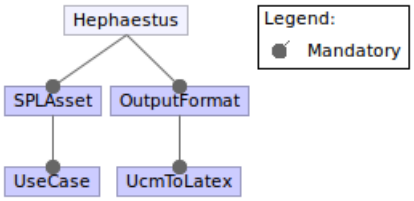

Figura 3.2: Configuração de Hephaestus na primeira versão.

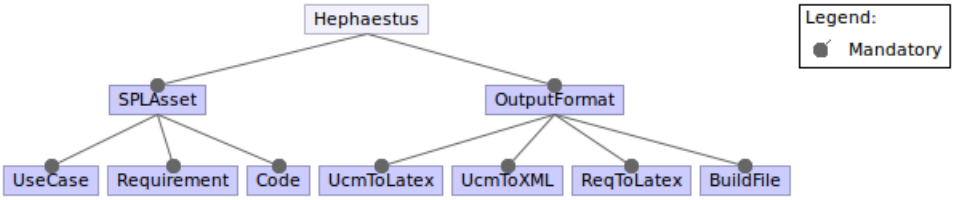

Figura 3.3: Configuração de Hephaestus na segunda versão.

Para atingir estes objetivos, novos tipos de dados e transformações foram necessárias, bem como parte do código existente teve que ser alterado. Precisamente, para introduzir suporte para variabilidades em requisitos de alto nível (ou apenas requisitos) e código-fonte nós tivemos que:

(a) implementar novos tipos de dados para representar os ativos requisitos e referências à código-fonte;

(b) implementar novas transformações para resolver variabilidades em requisitos e código-fonte. Transformações existentes variam de acordo com sua complexidade, cada uma gerando cerca de 10 a 100 linhas de código Haskell; e

(c) evoluir ambos os tipos de dados *SPLModel* e *InstanceModel*, bem como tivemos que rever tanto a função de exportação e a função *parser* XML do conhecimento da configuração, responsável pelo reconhecimento da sintaxe concreta das novas transformações (ver Figura 3.4).

Com base nos itens mencionados acima, a primeira conclusão é que, usando a arquitetura atual de Hephaestus, não é possível introduzir novos tipos de dados (representando a sintaxe abstrata de um novo ativo de LPS) e suas transformações de uma forma modular. Isso ocorre porque para introduzir suporte a um novo tipo de ativo temos que alterar o tipo de dados *SPLModel* e *InstanceModel*, o *parser* XML do conhecimento da configuração e a função *export* — ainda que o tipo de dados do conhecimento da configuração

```
1  data SPLModel =SPLModel {
2   ...
3       splReq  ::  RequirementModel,
4       splComponents :: ComponentModel
5  }
6
7  data InstanceModel =InstanceModel {
8   ...
9       req  ::  RequirementModel,
10      components :: ComponentModel,
11       buildEntries   ::  [ PreprocessingDirective ],
12       preProcessFiles   ::  [ PreprocessingFiles ]
13 }
14
15 exportProduct sourceDir  targetDir  product = do
16    exportUcmToLatex... (ucm product)
17    exportUcmToXML... (ucm product)
18    exportRequirementsToLatex... (req  product)
19    copySourceFiles ...  (components product)
20     exportBuildFile  ...  ( preProcessDirectives   product)
21     preprocessFiles  ...  ( preProcessFiles  p)
```

Figura 3.4: Tipos de dados *SPLModel* e *InstanceModel* após introduzir suporte à variabilidade em requisitos e código-fonte.

(*ConfigurationKnowledge*) e o interpretador (função *build*) apresentem algum grau de estabilidade pois não temos que mudar a sua implementação quando introduzimos suporte a variabilidade de novos ativos. Assim sendo, poderíamos dizer que os tipos de dados *SPLModel* e *InstanceModel* não são *abertos* visto que para introduzir novos ativos de LPS temos que alterar as definições desses tipos de dados.

Evoluir Hephaestus para suportar variabilidade em código-fonte apresenta uma questão interessante (ver Figura 3.4), uma vez que tivemos que introduzir um novo tipo de ativo (*splComponentes*) no *SPLModel*. Esse ativo é uma lista de pares de objetos que relaciona um nome à um caminho relativo de um arquivo de código-fonte. O mesmo tipo de ativo foi também introduzido no *InstanceModel*. Além disso, outros dois campos foram necessários no *InstanceModel*: (a) *buildEntries*, que declara diretivas de pré-processamento, e (b) *preProcessFiles*, que declara uma lista de arquivos que devem ser pré-processados. Estes campos são instanciados quando Hephaestus constrói um produto considerando as próprias transformações de uma configuração de produto.

O trecho de código na Figura 3.5 mostra o impacto sobre a função *parser* XML do conhecimento da configuração. A primeira versão de Hephaestus declara apenas as quatro primeiras sentenças do case na função *xml2Transformation* (de *selectScenarios* até *bindParameter*). A transformação *selectRequirement* trata a variabilidade em modelos de requisitos, enquanto que as restantes transformações resolvem a variabilidade em código-fonte. Poderíamos dizer que as funções *exportProduct* e *xml2Transformation* não são *abertas*, desde que novos formatos de saída e novos tipos de transformações não são introduzidos de uma forma modular em Hephaestus.

```
1  xml2Transformation :: XmlTransformation
2      → Parser Transformation
3  xml2Transformation transformation =
4  let
5    args =...
6    tnsName =xmlTransformationName transformation
7  in
8    case tnsName of
9    "selectScenarios" → Success ( selectScenarios  args)
10   "selectUseCases" → Success (selectUseCases  args)
11   "evaluateAspects" → Success (evaluateAspects  args)
12   "bindParameter" →...
13   "selectRequirements" → ...
14   "selectComponents" →...
15   "selectAndMoveComponent" →...
16   "createBuildEntries" → ...
17   "preprocessFiles" → ...
18   otherwise →Fail " ... "
```

Figura 3.5: Parte do código usado durante o *parser* XML do conhecimento da configuração.

3.2.2 Linha de Produtos Hephaestus

Embora adaptado para as necessidades da linha de produtos TaRGeT, alguns usuários desta versão de Hephaestus poderiam requerer configurações mais específicas da ferramenta. Por exemplo, alguns usuários poderiam estar interessados em gerenciar variabilidade apenas em requisitos e casos de uso; outros poderiam estar interessados em gerenciar variabilidade somente em código-fonte; e engenheiros TaRGeT deveriam estar interessados em gerenciar variabilidades em requisitos, casos de uso e código-fonte.

Além disso, novas extensões de Hephaestus foram recentemente propostas. Por exemplo, a versão atual de Hephaestus também suporta variabilidade em modelos de processos de negócio (Machado et al., 2011) e ativos Simulink. Novamente, para introduzir suporte a variabilidade para esses ativos nós implementamos novos tipos de dados para representar a sintaxe abstrata desses modelos, implementamos novas transformações para resolver suas variabilidades, evoluímos os tipos de dados *SPLModel* e *InstanceModel* e revisamos a função *parser* XML do conhecimento da configuração. Já a função *build* e outras funções de suporte e tipos de dados são compartilhados entre todas essas extensões e não sofreram alterações.

Portanto, existe uma quantidade significativa de comunalidade entre estas versões de Hephaestus. Além disso, a variabilidade existente em Hephaestus tem forma regular requerendo tipos de dados e funções "abertas", como já explicado anteriormente. A fim de explorar a comunalidade e gerenciar a variabilidade sistematicamente, torna-se essencial realizar o *bootstrapping* destas versões de Hephaestus para uma LPS – adiante denominada de Hephaestus-PL – e depois evoluí-la como uma LPS. O modelo de features pretendido para Hephaestus-PL está representado na Figura 3.6. As Seções seguintes exploram e comparam diferentes técnicas de desenvolvimento de LPS para o desenvolvimento de

Hephaestus-PL.

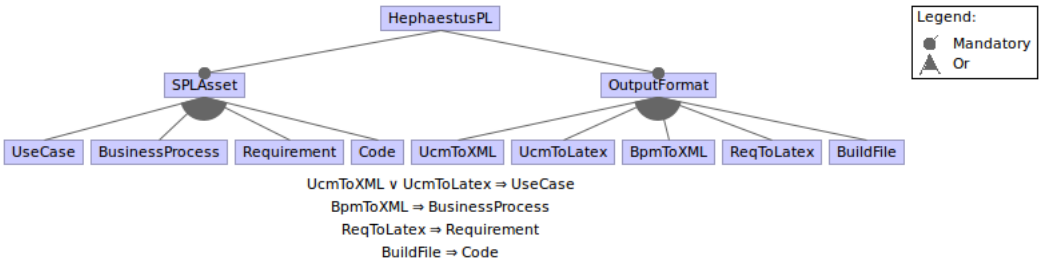

Figura 3.6: Modelo de Features da linha de produtos Hephaestus. Nesta versão, as *features* *SPLAsset* e *OutputFormat* definem um relacionamento *or-feature* com suas features filhas.

3.3 Processo de Avaliação Aplicado às Técnicas Analisadas

Nós aplicamos o método *Goal Question Metric* (GQM) (Basili et al., 1994) para ajudar a definir o contexto, o objeto de estudo, suas propriedades, o objetivo e como este último pode ser operacionalizado e respondido. Nesta seção discutimos os objetivos, questões e métricas (GQM) da nossa investigação e, então, apresentamos uma visão geral das técnicas avaliadas. A avaliação detalhada de cada técnica é apresentada na Seção 3.4.

3.3.1 Objetivos, Questões e Métricas

Este trabalho visa comparar a aplicabilidade de diferentes técnicas para o gerenciamento de variabilidades em LPS, em relação à fase de Engenharia do Domínio, para o desenvolvimento de produtos de software e no contexto das diferentes versões de Hephaestus, discutidas na seção anterior. De acordo com o GQM, a Tabela 3.1 resume o objetivo geral de nosso trabalho.

Propósito	comparar
Questão	aplicabilidade
Objeto	diferentes técnicas para gerenciar variabilidade
Ponto de Vista	perspectiva de engenharia de domínio
Contexto	diferentes versões de Hephaestus

Tabela 3.1: Objetivo GQM do artigo I.

Na Seção 3.2 caracterizamos o *contexto* e o *ponto de vista* do projeto inicial de Hephaestus que foi construído para gerenciar a variabilidade em cenários de casos de uso e sua evolução para suportar variabilidade em outros ativos. Na Seção 3.3.2 brevemente introduzimos o *objeto*, ou seja, os mecanismos avaliados e as técnicas relacionadas (linguagens e ferramentas). Depois disso, a Seção 3.4 prossegue com a análise detalhada de cada

técnica utilizando uma avaliação qualitativa que responde as métricas do GQM proposto, buscando alcançar o *propósito* e a *questão* do objetivo do estudo.

A partir do objetivo do estudo apresentado na Tabela 3.1, derivamos várias questões que melhor caracterizam o nosso estudo. Além disso, relacionado a cada questão, usamos uma ou mais métricas qualitativas para indicar o nível de conformidade das técnicas em relação ao objetivo do estudo. "Métricas" de avaliação qualitativa são também permitidas no método GQM. Esta investigação qualitativa é bem adequada pois permite avaliarmos atributos de alto nível que são relevantes para o projeto de Hephaestus-PL antes da sua efetiva implementação. Portanto, o uso de métricas quantitativas está fora do escopo deste trabalho mas deve ser contemplado em trabalhos futuros. A seguir, apresentamos as questões (**Q**) e as métricas (**M**) do nosso modelo GQM.

Q1 *A técnica tem expressividade para suportar a variabilidade de Hephaestus?*

- Métrica **M1.1** (Sim/Não): A técnica suporta tipos de dados "abertos".
- Métrica **M1.2** (Sim/Não): A técnica suporta funções "abertas".
- Métrica **M1.3** (Sim/Não): A técnica suporta a instanciação de um único ativo LPS.
- Métrica **M1.4** (Sim/Não): A técnica suporta a composição de ativos sem ter que instanciar todos os ativos definidos em Hephaestus-PL.

Q2 *O nível de granularidade suportado pela técnica para tratar variabilidade atende às necessidades de Hephaestus-PL?*

- Métrica **M2**: Nível de granularidade da técnica para tratar variabilidade.

Q3 *A técnica fornece gerenciamento modular dos ativos de Hephaestus-PL?*

- Métrica **M3** (Sim/Não): A técnica fornece suporte para modularização de features.

Q4 *Qual é o esforço para usar a técnica no código atual de Hephaestus para gerenciar a variabilidade em Hephaestus-PL?*

- Métrica **M4**: Nível de impacto do uso da técnica no código atual de Hephaestus.

Q5 *Qual é o nível de maturidade da técnica?*

- Métrica **M5.1**: Trabalhos relacionados, estudos de caso e aplicações que usam a técnica.
- Métrica **M5.2**: Domínios de negócio onde a técnica já foi aplicada.

Q6 *A técnica é aplicável na linguagem funcional Haskell?*

- Métrica **M6** (Sim/Não): A técnica poderia ser facilmente aplicada no código existente de Hephaestus escrito em Haskell.

Existem ainda outros fatores como tipo de feature, localização no código, *binding time*, espalhamento e interação da feature já utilizados em outros trabalhos mas que não fizeram parte do escopo do GQM deste artigo. Entretanto, o espalhamento da feature foi tratado no artigo apresentado no Capítulo 4 e o fator *binding time* não foi tratado porque é estático em Hephaestus-PL.

Questões Q1-Q6 representam características relevantes para o desenvolvimento de Hephaestus-PL para atingir os requisitos do projeto e minimizar o impacto sobre o sistema atual. Q1 foca no atributo mais importante da nossa avaliação que é a expressividade das técnicas para suportar a variabilidade de Hephaestus. Nesse sentido, as métricas M1.1, M1.2, M1.3 e M1.4 investigam se as técnicas analisadas suportam os tipos de variabilidade existentes em Hephaestus-PL, classificadas como tipos de dados "abertos", funções "abertas", instanciação de um único ativo e composição de ativos.

Em particular, a métrica M1.4 remete a um problema atual de Hephaestus: a necessidade de instanciar mais de um ativo LPS definido nos tipos algébricos *SPLModel* e *InstanceModel*. As respostas possíveis para essas métricas são "sim" ou "não". Q2 foca no nível de granularidade das técnicas para a implementação de variabilidades e sua aderência à implementação atual de Hephaestus. M2 pode ter como possíveis respostas "granularidade grossa" ou "granularidade fina". Q3 avalia se as técnicas suportam modularidade de features ou não, uma propriedade importante na abordagem reativa para o desenvolvimento de LPS, uma vez que isso facilita a inserção de novas features em Hephaestus e minimiza o conhecimento prévio dos detalhes dos módulos do sistema e a localização da implementação das features, que muitas vezes, estão espalhadas. As respostas possíveis para a métrica M3 são "sim" ou "não". Q4 foca no esforço necessário para aplicar cada técnica no código atual de Hephaestus. Portanto, M4 mede o nível de impacto das técnicas no código de Hephaestus. As respostas possíveis para M4 são "alto", "médio" ou "baixo". Q5 avalia o nível de maturidade das técnicas medindo o seu uso em trabalhos anteriores, estudos de casos e domínios de negócio. Valores possíveis de M5 são os nomes de trabalhos e domínios de negócios em que a técnica já foi aplicada. Finalmente, Q6 avalia a aderência das técnicas para a implementação atual de Hephaestus, mais especificamente, o uso da técnica em Haskell. As respostas possíveis para M6 são "sim" ou "não".

3.3.2 Mecanismos Avaliados

Kästner e outros autores discutem sobre duas formas comuns de implementar uma LPS: as abordagens composicional e anotativa (Kästner et al., 2008). No nosso estudo consideramos também mecanismos complementares baseados em transformação de programas e polimorfismo paramétrico. Estes são essenciais para ampliar o escopo do projeto e oferecer alternativas de implementação não restritas a determinados paradigmas (por exemplo, OO e AOSD). Nesta seção fazemos uma breve introdução a estas abordagens, deixando a apresentação dos detalhes de seu uso para resolver a variabilidade de Hephaestus PL para a Seção 3.4. Avaliamos a aplicação de cada abordagem usando, tanto a nível de implementação ou de projeto, linguagens específicas como Stratego/XT e Aspectual Caml, construtores de linguagens como tipo *class* em Haskell ou ferramentas como CIDE, que foram escolhidos considerando-se principalmente a nossa experiência e sua viabilidade em relação ao nosso domínio técnico que se traduz em extrair uma LPS a partir de um software desenvolvido em Haskell.

- **Abordagem Anotativa:** implementar as features de LPS usando abordagens anotativas exige alguma forma de anotação nos artefatos em que as features são projetadas. Anotações podem variar desde declarações `#ifdef` e `#endif` usadas em linguagens C ou C++ até tipos mais seguros (*type-safer*), mecanismos independentes de linguagem para implementar variabilidade como suportado pela ferramenta CIDE (Kästner et al., 2008, 2009) ou sua extensão semi-automatizada CIDE+ (Borges et al., 2010). Neste trabalho avaliamos o uso de CIDE para implementar variabilidade de Hephaestus, conforme apresentado na Seção 3.4.2.

- **Abordagem Transformacional:** resolver a variabilidade de LPS usando este mecanismo requer o desenvolvimento de linguagens de domínio específico e de transformações de código-fonte, que poderiam ser implementadas por linguagens e ferramentas como Stratego/XT (Visser, 2003). Neste trabalho avaliamos uma linguagem de domínio específico chamada Transkell, implementada usando Stratego/XT (Xavier e Borba, 2010) e projetada para resolver as variabilidades de Hephaestus, conforme descrito na Seção 3.4.3.

- **Abordagem Composicional:** nesta abordagem as features são implementadas como módulos distintos e uma instância da linha de produtos é gerada pela composição de um conjunto de módulos. Várias técnicas se enquadram nesta abordagem, tais como: tecnologias de componentes, *mixing layers*, AHEAD, separação multidimensional de interesses e programação orientada por aspectos (AOP). Neste contexto, AOP pode ser usada como uma abordagem composicional para a implementação de variabilidades em LPS. AOP visa uma melhor modularização de interesses transversais no desenvolvimento de aplicações onde as features são normalmente transversais (Mezini e Ostermann, 2004). Apesar da maioria das implementações estender linguagens orientadas à objeto, existem algumas implementações de AOP para linguagens funcionais, tais como Aspectual Camll (Masuhara et al., 2005) e AspectFun (Wang e Oliveira, 2009). Na Seção 3.4.1 avaliamos Aspectual Caml como uma técnica para gerenciar variabilidades em LPS usando o mecanismo composicional porque Aspectual Caml já está disponível na linguagem funcional Objective Caml e está mais estável do que AspectFun.

- **Polimorfismo Paramétrico:** polimorfismo paramétrico é um dos mecanismos que estende o poder de linguagens funcionais, como Haskell. Em Haskell, bem como em outras linguagens funcionais, as funções podem ser definidas como polimórficas em relação aos seus argumentos e tipos de retorno da função. Isto significa que uma função polimórfica poderia ser aplicada a qualquer tipo. No entanto, geralmente temos que expressar que uma função polimórfica deve ser aplicada somente à argumentos de um tipo T, sendo T uma instância de um tipo *class*. Na Seção 3.4.4 descrevemos a avaliação realizada com a técnica tipo *class* em Haskell para gerenciar as variabilidades das versões de Hephaestus.

3.4 Técnicas Avaliadas

Nesta seção, de acordo com o GQM apresentado na Seção 3.3.1, nós avaliamos nas perspectivas de projeto e implementação de uma linha de produtos de Hephaestus, as técnicas

associadas aos mecanismos de gerenciamento de variabilidades selecionados para avaliação no nosso trabalho e descritos na Seção 3.3.2.

As subseções seguintes apresentam para cada técnica: *(i)* a sua definição; *(ii)* a proposta de refatoração de Hephaestus em Hephaestus-PL usando a técnica, à nível de projeto ou implementação; e *(iii)* a avaliação da utilização da técnica no contexto de Hephaestus-PL.

3.4.1 Aspectual Caml

Aspectual Caml é uma linguagem de AOP derivada da linguagem funcional fortemente tipada chamada Objective Caml, que é um dialeto da linguagem funcional ML. Aspectual Caml estende o *parser* e o *type checker* do compilador Objective Caml, funcionando como um tradutor e dando suporte à transversalidade estática e dinâmica de features. Em particular, transversalidade estática permite estender tipos de dados através da adição de novos construtores em um tipo de dados e adição de novos campos em um construtor de um tipo de dados. Transversalidade dinâmica depende de *pointcuts* capturando eventos como chamadas de função e, em seguida, aplicando um trecho de *advice* com novo comportamento.

Refatorando Hephaestus em Hephaestus-PL usando Aspectual Caml

Para o desenvolvimento de Hephaestus-PL a partir do projeto e implementação atual de Hephaestus, é necessário refatorar os tipos de dados *SPLModel* e *InstanceModel* e o código da função *parser* XML do conhecimento de configuração, mostrado na Figura 3.5. Assim, os tipos de dados *SPLModel* e *InstanceModel* refatorados têm apenas o campo *FeatureModel*. Além disso, a função *xml2Transformation* refatorada não implementa o reconhecimento da sintaxe concreta das transformações das features (ver Figura 3.7). Isto é delegado a uma nova função *parser* que tem apenas o comportamento básico. O código refatorado representa a comunalidade da função *parser* a ser compartilhada entre as diferentes instâncias de Hephaestus-PL. Para cada *feature* variante em Hephaestus-PL, um aspecto é criado como segue: 1) um *advice* estenderá a função *parser* para implementar o reconhecimento da sintaxe concreta das transformações da feature; 2) transversalidade estática estenderá os tipos de dados *SPLModel* e *InstanceModel* para definir campo(s) adicional(is) correspondentes ao novo ativo(s) da *feature* variante. O código na Figura 3.8 ilustra a definição de um aspecto em Aspectual Caml, denominado "Aspecto UCM", que implementa a variabilidade associada à feature *UseCaseModel*.

Avaliação de Aspectual Caml

Os mecanismos de AOP em Aspectual Caml atendem aos principais requisitos para o desenvolvimento de Hephaestus-PL (Q1), ou seja, Aspectual Caml tem suporte para variabilidade de tipos de dados (tipos de dados "abertos") e funções (funções "abertas"). A composição de ativos é implementada pela composição de aspectos. O nível de granularidade suportado pela técnica é fino (Q2), porque permite o gerenciamento de variabilidade à nível de campo nos tipos de dados e isso está aderente ao espaço de variabilidade de Hephaestus que é, principalmente, fino. Aspectual Caml permite que as features sejam

```
xml2Transformation transformation =
let
  args =...
    tnsName =xmlTransformationName transformation
  in parse' tnsName args

parser' :: String → String → Parser Transformation
parser' "id" _ =Success Id
```

Figura 3.7: Refatoração do *parser* XML do conhecimento da configuração

```
aspect UCM {
type+ SPLModel =SPLModel of...∗UseCaseModel

type+ InstanceModel =
   InstanceModel of ... ∗UseCaseModel

pointcut pcUCMParser nameT argsT =
     call Parser' tnsName args

advice UCMParser =
     [around pcUCMParser tnsName args]

match tnsName with
     "selectScenarios"→Success ( selectScenarios args)
     "selectUseCases"→Success (selectUseCases args)
     "evaluateAspects"→Success (evaluateAspects args)
     "bindParameter"→...
}
```

Figura 3.8: Modularização da feature *UseCaseModel* usando Aspectual Caml

modularizadas (Q3) em aspectos que fornecem uma boa separação de interesses transversais, pois as extensões relacionadas com uma determinada feature variante estão confinadas a um único módulo e esse módulo aborda apenas as extensões relacionadas a esta *feature*. Com relação ao nível de maturidade (Q5), Aspectual Caml já foi aplicada na implementação de um protótipo de compilador de linguagem de programação. No entanto, Aspectual Caml não é aplicável aos códigos desenvolvidos em linguagem Haskell (Q6). Portanto, não é possível aplicar a técnica para o código atual de Hephaestus que foi desenvolvido em Haskell. Assim sendo, poderíamos sugerir duas alternativas para o uso dessa técnica em Hephaestus: *(i)* desenvolver uma extensão de Haskell para contemplar as funcionalidades de Aspectual Caml; ou *(ii)* traduzir o código Haskell de Hephaestus para Aspectual Caml. Em ambos os casos, o impacto sobre Hephaestus é alto (Q4).

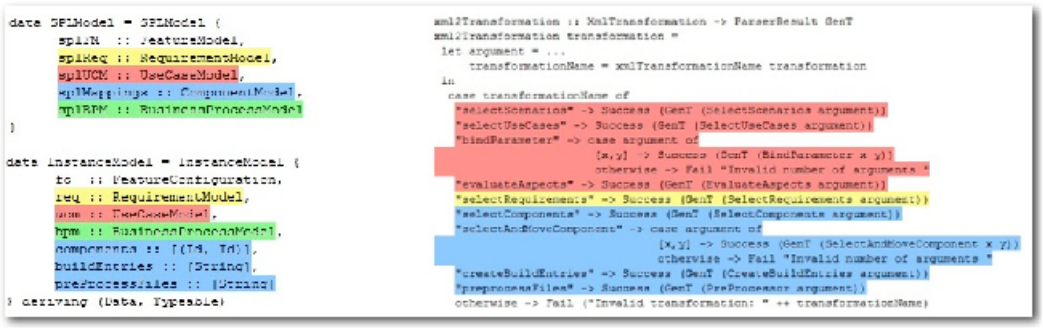

Figura 3.9: Parte do código de Hephaestus com as features anotadas usando CIDE.

3.4.2 CIDE

CIDE é uma ferramenta de abordagem anotativa para representar variabilidade em diferentes artefatos de LPS. Usando este ambiente, inicialmente, criamos um modelo de features e relacionamos cada feature com uma cor específica. Então, associamos trechos de código às correspondentes features no ambiente da ferramenta CIDE que irá mostrar os trechos de código "pintados" com a cor correspondente da feature (Kästner et al., 2008). Em seguida, podemos selecionar uma configuração de features e CIDE exportará um produto apenas com o código que implementa as features mandatórias e as opcionais selecionadas. CIDE implementa variabilidade em LPS sem ofuscar o código-fonte, diferente do que ocorre quando usamos diretivas de pré-processamento, a forma mais comum de abordagem anotativa. Por este motivo, alguns autores afirmam que CIDE emprega uma separação virtual de interesses e visões simplificadas do código.

Refatorando Hephaestus em Hephaestus-PL usando CIDE

O modelo inicial de features de Hephaestus-PL, apresentado na Figura 3.6, foi definido na ferramenta CIDE. A partir disso, começamos a associar e colorir os trechos de código relacionados a cada feature, que representam os ativos definidos no tipo de dados *SPLModel* e as features dos formatos de saída (ver Figura 3.9). Mais especificamente, foram coloridas as linhas de código das features contidas nos tipos de dados *SPLModel* e *InstanceModel*, na função *parser* XML do conhecimento da configuração, e na função *export*.

Avaliação de CIDE

Em relação à questão Q1, CIDE tem expressividade suficiente para suportar todos os tipos de variabilidade em Hephaestus. CIDE suporta variabilidade em tipos de dados, funções e a composição de ativos é garantida pelo processo de geração de uma variante que representa uma instância de produto que combina quaisquer ativos do Modelo de Features da LPS definido em CIDE. Dada a sua natureza anotativa, CIDE suporta variabilidade em ambos os níveis de granularidade: fino e grosso (Q2).

Como explicado anteriormente, CIDE fornece separação fraca de interesses ("separação virtual"). Por esta razão, nós não somos capazes de separar as projeções de features em módulos distintos, logo Q3 não é suportado por CIDE. Embora a ferramenta permita a visualização do código-fonte correspondente a uma feature selecionada, a implementação da mesma preserva um certo grau de espalhamento e entrelaçamento. Podemos ver isso observando as diferentes cores em tipos de dados específicos e funções na Figura 3.9. Por exemplo, usando CIDE, campos relacionados à diferentes ativos são ainda declarados nos mesmos tipos de dados *SPLModel* e *InstanceModel*. Isso pode trazer alguns problemas de modularidade quando tentarmos evoluir a linha de produtos, pois para introduzir um novo ativo, teremos que alterar o código de módulos existentes de Hephaestus-PL. Assim sendo, consideramos que CIDE não suporta o princípio *open-closed*, que recomenda que o projeto de software seja aberto para extensões mas fechado para modificações. No entanto, essa deficiência não é uma questão crítica em Hephaestus, porque o grau de espalhamento e entrelaçamento de features é consideravelmente pequeno e confinado a um número pequeno e constante de módulos a ser afetado pela adição de novas features.

Nós classificamos como médio o esforço para usar a técnica no código atual de Hephaestus (Q4). Considerando os artefatos a serem anotados, a atividade de colorir os trechos de código associados às features não demanda muito tempo. Além disso, esse esforço pode ser minimizado e tornar-se baixo se for utilizado uma abordagem similar a CIDE+, uma extensão de CIDE que permite anotação (coloração) semi-automática de trechos de código através da definição da "semente" da feature. Em relação ao nível de maturidade de CIDE (Q5), há uma variedade de publicações sobre o uso de CIDE, por exemplo, linguagens de programação, estudos de casos de banco de dados e sistemas operacionais. O uso de CIDE+ tem sido relatado em *bootstrapping* de LPS para ferramentas CASE (Couto et al., 2011). Como CIDE suporta Haskell então é possível refatorar Hephaestus usando CIDE, assim sendo, a técnica foi avaliada positivamente em Q6.

3.4.3 Transkell

Transkell é uma linguagem de domínio específico (DSL) desenvolvida para gerenciar a variabilidade de Hephaestus (Xavier e Borba, 2010). Transkell é a primeira tentativa de evoluir Hephaestus para uma linha de produtos, mesmo sabendo que sua implementação atual não suporta toda a variabilidade descrita aqui neste artigo. Transkell foi projetada para simplificar o processo de inserção de novos tipos de transformações em Hephaestus. A Figura 3.10 mostra um exemplo concreto de uma transformação em Transkell.

Nossa proposta tem uma perspectiva diferente, uma vez que os ativos de LPS são nossa principal estratégia de decomposição do modelo de features, e não as transformações consideradas em Transkell.

Refatorando Hephaestus em Hephaestus-PL usando Transkell

Para criar uma versão específica de Hephaestus usando esta abordagem, engenheiros de produto escrevem um programa em linguagem Transkell e, em seguida, aplicam transformações de código Transkell para código Haskell, implementadas em Stratego/XT (Visser, 2003), traduzindo o programa-fonte em Transkell para código Haskell.

Um programa Transkell compreende um conjunto de transformações onde cada transformação especifica o que é adicionado ao código base quando uma transformação for

```
Transformation SelectComponents {
  BasicType {...}
  Input  {...}
  Output {
    Import (...)
    Code {
      exportUcmToLatex f ucm =...
      exportUcmToXML f ucm =...
    }
    ExportCode {...
     exportUcmToLatex... (ucm product)
     exportUcmToXML... (ucm product)
    }
  }
  SPL {
...
   type {
     ucm =UseCaseModel
   }
  }
}
```

Figura 3.10: Código Transkell que implementa a transformação *selectComponents*.

selecionada e executada. Por exemplo, uma transformação que seleciona um ativo da LPS deve especificar as declarações de código-fonte que serão adicionadas nos tipos de dados *SPLModel* e *InstanceModel*, na função *parser* XML do conhecimento da configuração, na função *parser* de leitura do ativo, na função *export* e, assim por diante.

Avaliação de Transkell

Com base na atual implementação de Transkell percebemos que um abordagem transformacional com uma linguagem de domínio específico implementada em Stratego/XT é capaz de gerenciar todas as variabilidades de Hephaestus (Q1), atendendo ambas as granularidades fina e grossa (Q2).

Além disso, as decisões de projeto de Transkell apresentam algum grau de modularidade (Q3), já que os desenvolvedores puderam descrever tudo relacionado a uma transformação dentro de um único construtor da linguagem Transkell. No entanto, consideramos que implementar uma linguagem de domínio específico para Hephaestus usando um modelo de features baseado nos ativos de LPS como requisitos, casos de uso e processos de negócios em vez de um modelo de features baseado em transformações, poderia resultar em um projeto mais modular. A principal razão é que, baseado nas decisões de projeto atuais de Transkell, sempre que duas transformações (T_1 e T_2) exigem contribuições semelhantes no produto final, temos que duplicar código ou criar dependências entre as transformações T_1 e T_2.

Em relação à Q4, o *bootstrapping* da linha de produtos a partir do código existente de Hephaestus requer a reescrita, usando a linguagem Transkell, de todos os módulos Haskell

que apresentam alguma variabilidade. Portanto, a fim de refatorar Hephaestus em uma linha de produtos usando uma DSL, um esforço substancial é necessário. No entanto, Marcos e Borba relatam sobre a maturidade de ferramentas para Stratego/XT (incluindo Spoofax) que tornam mais fácil (Q5) a construção da linguagem Transkell (Xavier e Borba, 2010) e as transformações de código para código entre Transkell e Haskell (Q6).

3.4.4 Tipo Classe

A ideia é criar transformações genéricas para Hephaestus, por exemplo, uma transformação *select asset* genérica usando a declaração tipos de classes de Haskell para resolver variabilidade na linha de produtos Hephaestus. Em sua forma básica de utilização, um tipo *class* permite aos desenvolvedores definir funções que são aplicadas a alguns, mas não a todos os tipos.

Um uso mais avançado de tipo *class* (Jones, 1995) tem sido usado para implementar tipos de dados e funções "abertas" em linguagens funcionais (Lämmel e Ostermann, 2006), duas classes de variabilidades encontradas em Hephaestus (ver Seção 3.2).

Refatorando Hephaestus em Hephaestus-PL usando Tipo Classe

A implementação de transformações genéricas em Hephaestus usando tipos de classes envolve duas etapas. Em primeiro lugar, a definição de tipos de classes para permitir a implementação de transformações genéricas (tais como *select asset*, *bind parameter* e *evaluate advice*). Note que as transformações para a seleção de ativos (*select asset*) são disponíveis para todos os modelos de ativos LPS (requisitos, cenários de casos de uso, processos de negócios, componentes e assim por diante) de Hephaestus. Já as implementações para as transformações *bind parameter* e *evaluate advice* são apenas disponíveis para modelos de ativos como cenários de casos de uso e processos de negócios. Em segundo lugar, temos que especificar de quais tipos de classes um modelo de ativo é instância de classe. Por exemplo, o modelo de caso de uso deve ser uma instância de todos os tipos de classes requeridos pelas transformações genéricas *select asset*, *bind parameter* e *evaluate advice*; enquanto o modelo de componentes deve ser uma instância apenas do tipo de classe *select asset*.

Esta solução utiliza construções avançadas de tipos, como é possível perceber observando o trecho de código na Figura 3.11, que declara três tipos de classes para generalizar a transformação *select asset*. Primeiro, declaramos um tipo classe |Id| para valores que podem ser usados para identificar ativos. Isso é necessário porque o tipo de um valor usado para identificar os ativos pode variar de acordo com um específico modelo de LPS. Uma instância do tipo classe |Id| deve ser também uma instância do tipo classe |Eq| (veja a restrição definido |Eq => Id x|). Além disso, nós declaramos um tipo classe |*Identifiable*| que declara uma função para obter a identidade de um elemento (no nosso caso, um ativo LPS). Esta classe é parametrizada de acordo com o tipo de um elemento identificável por |x| e o tipo de sua identidade |i|, onde |i| deve ser uma instância do tipo classe |Id|. Com isto, nós estabelecemos uma relação de tipos entre elementos identificáveis e identidades. Finalmente, um tipo classe |Composite| foi declarado para obter os componentes de um ativo (por exemplo, os requisitos de um modelo de requisitos) e substituir de alguma forma os componentes de um ativo.

```
class Eq x ⇒ Id x

class Id i ⇒ Identifiable x i | x → i
 where
  identify :: x → i

class (Eq y, Identifiable y i) ⇒ Composite x y i | x → y i
 where
  toComponents :: x → [y]
  replaceComponents :: ([y] → [y]) → x → x

selectAssets ::
  (Composite (f a) c i,
   Composite (g a) c i,
   Composite a c i,
    Identifiable c i, Id i
  ) ⇒ [i] →f a → g a → g a

selectComponents ids spl product =...
```

Figura 3.11: Hierarquia de tipos de classes para gerar a transformação *select asset*.

Com os tipos de classes definidos acima, é possível implementarmos uma função genérica |selectAssets| que espera três parâmetros:

- Uma lista de identidades |[i]|.
- Um ativo LPS |f a|, o qual pode incluir um modelo de casos de uso, um modelo de requisitos, e assim por diante.
- Um produto de ativos |g a|, o qual pode incluir um modelo de casos de uso, um modelo de requisitos, e assim por diante. Mas note que, se o segundo argumento compõe um modelo de casos de uso, o produto resultante deverá também incluir um modelo de casos de uso (ambos |f a| e |g a| referem-se ao mesmo tipo de parâmetro |a|).

Como foi explicado, depois de declararmos os tipos classes necessários e as transformações genéricas, temos que definir quais ativos específicos devem ser instâncias do tipo classe |Composite|, respeitando todas as restrições e fornecendo implementações para ambas as funções |toComponents| e |replaceComponents| do tipo classe |Composite|.

Avaliação para Tipo Classe

Após o esforço inicial para generalizar algumas das transformações de Hephaestus, percebemos que usando somente tipo classe não seríamos capazes de derivar algumas instâncias de Hephaestus-PL. Por exemplo, usando o mecanismo explorado aqui, somos capazes de derivar produtos que suportam variabilidade em um único tipo de ativo. Isso não representa uma limitação impeditiva do uso da técnica, pois acreditamos que a maioria dos

Abordagens			Anotativa	Transformacional	Composicional	Refatoração
Técnicas			CIDE	Transkell	Aspectual Caml	Tipo Classes
G	Q	M				
G1	Q1	M1.1	Sim	Sim	Sim	Sim
		M1.2	Sim	Sim	Sim	Sim
		M1.3	Sim	Sim	Sim	Sim
		M1.4	Sim	Sim	Sim	Não
	Q2	M2	Fina/Grossa	Fina/Grossa	Fina	Fina
	Q3	M3	Não	Sim	Sim	Sim
	Q4	M4	Médio/ Baixo(CIDEPlus)	Alto	Alto	Alto
	Q5	M5.1	DB/OS/ ArgoUML/CIDEPlus	vários	compiladores	vários
		M5.2	engenharia de software	vários	linguagens de programação	vários
	Q6	M6	Sim	Sim	Não	Sim

Tabela 3.2: Resumo da avaliação das técnicas analisadas.

usuários de Hephaestus estão interessados em gerenciar variabilidades em modelos específicos de ativos. No entanto, tal solução não atenderia a necessidade da equipe do TaRGeT que precisa gerenciar variabilidades em requisitos, casos de uso e modelos de componentes. Portanto, com relação à expressividade da técnica (Q1), o nosso projeto inicial usando tipo de classes não atende aos requisitos de Hephaestus-PL, mas especificamente não atende à métrica M1.4 (composição de ativos).

Com relação ao nível de granularidade (Q2), o uso de tipo classe, como descrito aqui, suporta tanto variabilidade grossa e fina, de uma forma modular (Q3), uma vez que somos capazes de estender os tipos de dados *SPLModel* e *IntanceModel* com novos ativos, declarando novas instâncias dos tipos classes existentes. No entanto, para introduzir suporte à variabilidade nas funções *export* e *xml2Transformation* para os novos ativos, provavelmente, temos que definir outros tipos de classes ou combinar tipos de classes com o conceito de *monads* em Haskell, algo que requer uma profunda investigação e será abordado em trabalhos futuros.

O esforço (Q4) para refatorar Hephaestus em Hephaestus-PL usando tipo classes foi considerado alto, pois a maioria do código existente descrito aqui, teria que ser reescrito. Além disso, o uso desta técnica requer um conhecimento mais profundo sobre os conceitos e utilização de tipo classe e sistemas de tipos em Haskell, não sendo facilmente acessível a alguns desenvolvedores que contribuem no projeto Hephaestus. No entanto, tipo classes têm sido profundamente discutido (Q5) em comunidades de linguagens funcionais com muitas aplicações descritas — principalmente relacionadas com o conceito de funções e tipos de dados "abertos". A técnica é totalmente implementada no compilador GHC (*Glasgow Haskell Compiler*) utilizando algumas extensões de tipos do Haskell.

3.4.5 Síntese das Técnicas Avaliadas

Nesta seção apresentamos uma visão geral das técnicas com relação ao modelo GQM proposto e discutimos as técnicas mais apropriadas para o projeto e implementação de Hephaestus-PL, considerando o código atual de Hephaestus. A Tabela 3.2 apresenta a síntese da avaliação de cada técnica por cada métrica do modelo GQM.

Em relação à expressividade das técnicas para tratar os tipos de variabilidade em Hephaestus, observamos que todas as técnicas atendem à este critério, exceto a técnica *Tipo Classe* que não suporta composição de ativos da LPS (M1.4). Em Hephaestus, o escopo de variabilidade é, na maior parte, de granularidade fina, e para esse critério todas as técnicas atendem e poderiam ser aplicadas ao desenvolvimento de Hephaestus-PL. No entanto, o critério de suporte à modularização para facilitar a manutenibilidade ao introduzir novos ativos de LPS em Hephaestus-PL, não é atendido apenas pela técnica *CIDE* que é uma abordagem anotativa. Em geral, o nível de impacto sobre o código de Hephaestus na aplicação da técnica foi considerado baixo quando a técnica suporta a linguagem Haskell, como ocorre com a ferramenta CIDE. Entretanto, no caso da técnica *Tipo Classe*, mesmo sendo suportada por Haskell, o impacto é alto conforme justificado na avaliação dessa técnica. Para a técnica *Transkell*, o impacto é alto porque nós teríamos que reescrever todos os módulos Haskell de Hephaestus que apresentam alguma variabilidade usando a linguagem Transkell. A técnica *Aspectual Caml* também foi avaliada como alto impacto em Hephaestus, principalmente porque essa técnica não oferece suporte à linguagem Haskell. Sobre o critério nível de maturidade das técnicas, todas atendem porque há várias obras já implementadas com as referidas técnicas. Finalmente, quase todas as técnicas avaliadas apresentam suporte à linguagem funcional Haskell usada na implementação de Hephaestus, com exceção da técnica Aspectual Caml que suporta apenas a linguagem funcional Objective Caml.

Nossa avaliação final é que as técnicas Aspectual Caml e Tipo Classe são as que apresentaram as métricas mais desfavoráveis (no caso, duas medidas) para os requisitos definidos de projeto e implementação de Hephaestus-PL. Por outro lado, identificamos que as técnicas CIDE e Transkell são as mais adequadas para o desenvolvimento de Hephaestus-PL a partir do código atual de Hephaestus. CIDE, apesar de ser uma técnica de abordagem anotativa e não suportar a modularização física das features (ativos), consideramos-a adequada porque o nível de espalhamento e entrelaçamento do código das features é baixo em Hephaestus. A técnica Transkell, apesar de ter sido avaliada com alto impacto (M4) para o código atual de Hephaestus, de acordo com a Seção 3.4.3, Marcos e Borba relatam sobre a maturidade de ferramentas para Stratego/XT que facilitam a construção de uma linguagem de domínio específico como Transkell e suas transformações de código-fonte de Transkell para Haskell.

Sobre a completude das técnicas avaliadas com relação aos requisitos de Hephaestus-PL, concluímos que nenhuma técnica isolada é capaz de atendar positivamente todas as métricas definidas no modelo GQM. Portanto, uma possível estratégia seria combinar abordagens buscando essa completude no atendimento das métricas. Em particular, sugerimos combinar as duas técnicas mais bem avaliadas, ou seja, CIDE e Transkell. Isso é viável por causa da disponibilidade imediata dessas ferramentas para Haskell. Além disso, a fim de resolver a deficiência de CIDE com relação à modularidade, os artefatos contendo ativos com variabilidade de granularidade fina poderiam ser manipulados pelas transformações de Transkell, já que são poucos em Hephaestus e o uso de Transkell seria mínimo,

favorecendo a métrica M4 de Transkell para Hephaestus-PL. Para outros artefatos contendo ativos com granularidade média ou grossa, poderia ser aplicado CIDE ou CIDE+, considerando o baixo esforço de sua aplicação sem comprometer a modularidade. A implementação dessa estratégia combinada poderia ser iniciada com o próprio Hephaestus, aproveitando as funções *build* e outras funções de apoio, tipos de dados e as transformações que tratam a seleção de ativos (*selectAsset*) e variabilidade anotativa (*evaluateAdvice*). Nós consideramos a implementação dessa estratégia como trabalho futuro.

3.5 Trabalhos Relacionados

Diferentes técnicas de projeto e implementação de LPS têm sido comparadas (Anastasopoulos e Gacek, 2001; Svahnberg et al., 2005; Lopez-Herrejon et al., 2005). No entanto, isso não têm sido realizado no contexto de ferramentas de desenvolvimento de LPS, como fizemos neste trabalho. O trabalho anterior (Batory et al., 2003) considerou que uma técnica composicional, no caso a programação orientada por aspectos (AOP) implementada em AHEAD, poderia realizar o *bootstrapping* da ferramenta em si, portanto, potencialmente levando a uma família de ferramentas AHEAD. No entanto, isso não tem sido explorado para tratar a variabilidade em diferentes artefatos como proposto aqui. Por outro lado, as abordagens anotativas CIDE e CIDE+ podem tratar variabilidade em diferentes artefatos. No entanto, diferentemente do nosso trabalho, nenhuma das abordagens mencionados anteriormente trata a combinação de diferentes artefatos para atender *or-features* (Figura 3.6). Da mesma forma a este trabalho, Apel et al.l (Apel et al., 2009b) compararam uma abordagem anotativa em relação à composicional visando a modularização de features. Segundo seu estudo, cada uma traz benefícios relativos e concluíram que a combinação das abordagens é mais adequada para modularização de features. A abordagem sinergética é semelhante a nossa conclusão, mas nós consideramos outras abordagens na avaliação como a transformacional e polimorfismo paramétrico, e concluímos que a abordagem anotativa é melhor se combinada com a transformacional. Além de Hephaestus, existe uma infinidade de outras ferramentas de derivação de produtos de LPS como, por exemplo, *pure::variants*, *GenARch* e *Gears*. Entretanto, como dito em outros trabalhos (Rabiser et al., 2010), a maioria dessas ferramentas carece de flexibilidade e adaptabilidade.

3.6 Considerações Finais

Efetuamos uma análise comparativa de técnicas de gerenciamento de variabilidades para o desenvolvimento de ferramentas de LPS no contexto da ferramenta Hephaestus. A análise revelou que duas técnicas, uma anotativa (CIDE) e outra transformacional (Transkell), são mais adequadas para o gerenciamento da variabilidade em Hephaestus e que a sua combinação é uma estratégia viável para melhorar esse gerenciamento. Embora, o estudo comparativo ocorreu no contexto de apenas uma ferramenta, no caso Hephaestus, acreditamos que a análise pode ser útil para o desenvolvimento de outras ferramentas, uma vez que estas podem ter abstrações e variabilidades semelhantes aos tipos de dados e funções de Hephaestus, e assim serem tratadas pelas técnicas discutidas aqui. No entanto, mais trabalhos empíricos são necessários para validar o que foi discutido aqui. Como traba-

lho futuro estamos planejando implementar a estratégia descrita na Seção 3.4.5 para o *bootstrapping* de Hephaestus- PL. A implementação focará no reuso e generalização das transformações e na garantia de *type safety* de produtos instanciados de uma forma escalável. Nós também planejamos conduzir mais estudos empíricos avaliando a evolução de Hephaestus-PL para tratar a variabilidade em diferentes tipos de artefatos.

Capítulo 4

Artigo II - Suporte à Configurabilidade e Flexibilidade em Desenvolvimento de Ferramentas de Linha de Produtos de Software: o estudo de caso Hephaestus-PL

Este capítulo corresponde ao Artigo II intitulado **Suporte à Configurabilidade e Flexibilidade em Desenvolvimento de Ferramentas de Linha de Produtos de Software: o estudo de caso Hephaestus-PL** escrito por Lucinéia Turnes da Universidade de Brasília, Ralf Lämmel da Universität Koblenz-Landau (Alemanha), Vander Alves e Rodrigo Bonifácio, ambos também da Universidade de Brasília.

As Seções 4.1 e 4.2 deste Artigo II apresentam certa semelhança com relação à descrição do contexto do problema e sua motivação relatados nas Seções 3.1 e 3.2 do Artigo I descrito no Capítulo 3, pois ambos os artigos focam na necessidade de melhorar a configurabilidade e flexibilidade de Hephaestus.

Resumo

Suporte ferramental é essencial para a Engenharia de Aplicação em Linhas de Produto de Software. Apesar de uma variedade de ferramentas existentes, na maioria delas falta suporte adequado à configurabilidade e flexibilidade. Nesse caso, é difícil para elas serem aplicadas em diferentes contextos, por exemplo, endereçar variabilidade em uma combinação arbitrária de diferentes artefatos e introduzir e gerenciar variabilidades em novos artefatos. Para abordar esta questão é necessário explorar sistematicamente a comunalidade e adequadamente gerenciar a variabilidade de tais ferramentas. Assim sendo, este artigo apresenta a análise de domínio, projeto e implementação de Hephaestus-PL, uma linha de produtos de software de ferramentas de linha de produtos de software, e um processo que suporta a extensão de Hephaestus-PL. Hephaestus-PL é suportado por um processo que permite instanciar ferramentas de linha de produtos para modelar variabilidades em novos e em qualquer combinação de artefatos e, foi desenvolvido pelo *bootstrapping* de versões

anteriores da ferramenta Hephaestus. Uma avaliação revela que Hephaestus-PL melhorou a configurabilidade e flexibilidade quando comparado à evolução anterior de Hephaestus.

4.1 Introdução

Linha de Produtos de Software (LPS) é um conjunto de sistemas de software que compartilham um conjunto de features comuns e gerenciadas que satisfazem as necessidades específicas de um determinado segmento de mercado ou missão e que são desenvolvidas a partir de um conjunto comum de ativos principais (core assets) de uma forma pré-definida (Clements e Northrop, 2001). Os benefícios potenciais incluem a melhoria da produtividade com menores custos de desenvolvimento e de tempo de lançamento no mercado e aumento da qualidade. Para alcançar esses benefícios, o suporte ferramental para realizar as atividades subjacentes desse processo é essencial. Em particular, o suporte ferramental apoia a Engenharia de Aplicação em que um produto é definido pela seleção de um grupo de features e, então, é realizada uma composição cuidadosamente coordenada das partes de diferentes componentes envolvidos. Devido à inerente complexidade e a necessária coordenação no processo de derivação (Griss, 2000), esta atividade é lenta e propensa a erros. Como resultado, a derivação de produtos individuais a partir de ativos de software compartilhados é ainda uma atividade demorada e cara em muitas organizações (Deelstra et al., 2005).

Conforme relatado por uma Revisão de Literatura Sistemática contemporânea que contou com análises de especialistas (Rabiser et al., 2010), requisitos chaves de ferramentas de derivação de produtos são configurabilidade e flexibilidade, que o mesmo estudo identifica como uma deficiência da maioria das ferramentas existentes. De fato, estas devem ser adaptadas à diferentes contextos, por exemplo, tratar com qualquer combinação de diferentes artefatos. Por exemplo, cada diferente configuração da ferramenta poderia suportar uma combinação específica de artefatos processos de negócio, código e requisitos. Além disso, a mudança das necessidades dos usuários e a evolução contínua da linha de produtos motivam ainda mais a flexibilidade e configurabilidade de ferramentas de derivação do produtos para atender às necessidades futuras, por exemplo, tratar com novos artefatos tais como modelos arquiteturais. E, sem um processo que suporte a inserção de novos artefatos, o tratamento com tais dispositivos de variabilidade não é efetivo.

Portanto, para fornecer configurabilidade e flexibilidade, torna-se necessário gerenciar adequadamente a variabilidade dentro dessas ferramentas tratando as próprias ferramentas como LPS, como também foi sugerido por Grünbacher et al. (Grünbacher et al., 2008). Apesar do esforço existente nessa direção (Vierhauser et al., 2011; Grünbacher et al., 2008; Batory et al., 2003), o gerenciamento de variabilidades em novos diferentes artefatos e em uma combinação qualquer de artefatos ainda não está plenamente atendido. Além disso, o detalhamento da análise de domínio, projeto e implementação e um processo que suporta a evolução da LPS não são fornecidos nem são publicamente disponíveis.

Nesse sentido, as contribuições apresentadas neste capítulo são:

- análise de domínio, projeto e implementação de Hephaestus-PL, uma linha de produtos de software flexível e configurável de ferramentas de linha de produtos de software;
- um processo para evoluir Hephaestus-PL endereçando novos artefatos.

A primeira contribuição trata a viabilidade de endereçar o problema. Em particular, Hephaestus-PL está publicamente disponível e foi desenvolvido para gerenciar comunalidade e variabilidade a partir das variantes existentes de Hephaestus (Bonifácio et al., 2009), uma ferramenta de LPS desenvolvida em Haskell e originalmente destinada ao gerenciamento da variabilidade em cenários de casos de uso, mas que evoluiu para tratar variabilidade independentemente em diferentes tipos de artefatos. Uma descrição detalhada de Hephaestus incluindo cenários de uso pode ser encontrada em (Bonifácio et al., 2009). O projeto de Hephaestus-PL está centrado no conceito de *bootstrapping* – com um *kernel* representando os elementos chaves comuns das variantes existentes de Hephaestus – e cujo gerenciamento de variabilidades foi implementado usando metaprogramação em Haskell. A segunda contribuição refere-se à eficácia de endereçar o problema: Hephaestus-PL é suportado por um processo que permite instanciar ferramentas de linha de produtos para modelar variabilidades em novos artefatos (por exemplo, processos de negócio, cenários de casos de uso e código). Uma avaliação revela que Hephaestus-PL melhorou a configurabilidade e flexibilidade quando comparado à evolução anterior de Hephaestus. Nós acreditamos que estas contribuições poderiam ser alavancadas para outros contextos em que a configurabilidade e flexibilidade de linhas de produtos sejam requisitos chaves.

O restante deste capítulo está organizado como segue. A Seção 4.2 descreve brevemente Hephaestus e sua evolução para endereçar diferentes artefatos. Em seguida, baseado em tais versões de Hephaestus, as Seções 4.3 e 4.4 apresentam a análise de domínio e projeto de Hephaestus-PL, respectivamente, enquanto que sua implementação está explicada na Seção 4.5. Seção 4.6 apresenta um processo que suporta a extensão de Hephaestus-PL e, a Seção 4.7 fornece uma avaliação e discussão de Hephaestus-PL. Trabalhos relacionados estão descritos na Seção 4.8 e a Seção 4.9 apresenta as observações finais.

4.2 Hephaestus

Hephaestus (Bonifácio et al., 2009) é uma ferramenta de derivação de produtos (Deelstra et al., 2005) disponível publicamente[1] que recebe contribuições de diferentes instituições de ensino: Universidade Federal de Pernambuco, Universidade de São Paulo e Universidade de Brasília. Inicialmente desenvolvido como uma ferramenta de prova de conceito para o gerenciamento de variabilidades em cenários de casos de uso (Bonifácio e Borba, 2009), Hephaestus fornece uma especificação declarativa e executável em Haskell de uma abordagem composicional (Kästner et al., 2008) e paramétrica para gerenciar variabilidade em cenários de casos de uso. Atualmente, Hephaestus suporta variabilidade em diferentes tipos de artefatos, que vão desde processos de negócios e modelos Simulink até código-fonte, e Hephaestus tem sido usado como a ferramenta de derivação em uma linha de produtos industrial (Ferreira et al., 2010).

Para o propósito inicial da ferramenta foi implementado primeiro:

- tipos de dados específicos para representar modelo de casos de uso (UCM), modelo de features (FM) e modelo de conhecimento da configuração (CK) (Czarnecki e Eisenecker, 2000), que relaciona expressões de features em lógica proposicional à transformações que tratam com a variabilidade em casos de uso;

[1] http://bit.ly/iRMMZM

- funções específicas que resolvem variabilidades em cenários de casos de uso, selecionando casos de uso ou cenários a partir de um modelo de LPS, compondo-os e resolvendo os parâmetros de acordo com as configurações de features específicas. Além disso, Hephaestus fornece uma função `build` que se comporta como um interpretador para o conhecimento da configuração (CK) e é responsável pela construção de um produto específico dada uma seleção de features, ou seja, uma configuração de features.

A Figura 4.1 apresenta um trecho de código da implementação inicial de Hephaestus destacando os tipos de dados relacionados ao conhecimento da configuração (linhas 1-5) e um interpretador correspondente (a função `build` nas linhas 7-17) e a assinatura das funções de transformação (linha 22). O conhecimento da configuração relaciona o espaço do problema através de uma expressão de features ao espaço da solução através de uma lista de transformações de artefatos. Cada transformação resolve uma parte da variabilidade nos artefatos reusáveis da LPS. A função `build` utiliza quatro artefatos de entrada (FM, FC, CK e artefatos reusáveis da LPS) para gerar uma instância de produtos. O processo *build* de derivação de produtos em Hephaestus avalia as linhas do CK validando cada expressão de features de acordo com o FC — o conjunto de features que caracteriza um determinado produto. Se uma expressão de features do CK é verdadeira para um dado FC então as transformações correspondentes são aplicadas ao produto sendo gerado. Além disso, este trecho de código da Figura 4.1 também mostra a representação inicial dos tipos de dados `SPLModel` (linhas 24-27) e `InstanceModel` (linhas 29-32). `SPLModel` empacota os artefatos cujo gerenciamento de variabilidades está sendo endereçado pela ferramenta Hephaestus e sua variabilidade está descrita no campo `splFeatureModel`, enquanto que `InstanceModel` empacota os artefatos depois que uma transformação foi aplicada a eles reduzindo a sua variabilidade. Eventualmente, toda a variabilidade é removida e o valor do `InstanceModel` corresponde ao campo configuração de features. A função `exportProduct` (linhas 34-36) gera uma representação em LaTeX de um produto específico de modelo de caso de uso.

4.2.1 Evolução de Hephaestus

No contexto de um projeto *R&D*, Hephaestus foi utilizado para substituir uma ferramenta proprietária que era usada para gerenciar variabilidades em uma linha de produtos industrial (Ferreira et al., 2010), em que novos artefatos de produtos deveriam ser exportados como uma consequência do processo `build`. Hephaestus então evoluiu passando a gerenciar variabilidades não apenas em cenários de casos de uso, mas também em requisitos de alto nível e código-fonte. Este último pela seleção de arquivos específicos que deveríam ser compilados bem como por iniciar um pré-processamento para resolver variabilidade em código-fonte. Apresentamos a configuração de ambas as versões nas Figuras 4.2 e 4.3.

De modo a alcançar estes objetivos, novos tipos de dados e transformações foram necessários, bem como parte do código existente de Hephaestus teve que ser alterado. Precisamente, para introduzir suporte às variabilidades em requisitos de alto nível (ou apenas requisitos) e código-fonte foi necessário:

(a) implementar novos tipos de dados para representar os ativos requisitos e referências à código-fonte;

```
type ConfigurationKnowledge =[ConfigurationItem]
data ConfigurationItem = ConfigurationItem {
   expression  = FeatureExpression ,
   transformations  = [Transformation]
}

build  ::  FeatureModel
       →  FeatureConfiguration
       →  ConfigurationKnowledge
       →  SPLModel
       →  InstanceModel
build  fm fc ck spl  = derive  ts  spl  emptyInstance
where
     ts  =concat  [transformations  c  |  c  ∈  ck,  eval  fc  (expression  c)]
     ucmodel        =splUCM spl
     emptyUCM     =ucmodel { useCases = []  ,  aspects  = []  }
     emptyInstance =InstanceModel fc  emptyUCM

derive  []  spl  product =product
derive  (t:ts)  spl  product = derive  ts  spl  (t  spl  product)

type Transformation =SPLModel →InstanceModel →InstanceModel

data SPLModel =SPLModel {
   splFeatureModel  ::  FeatureModel,
   splUCM :: UseCaseModel
}

data InstanceModel =InstanceModel {
    featureConfiguration   ::  FeatureConfiguration ,
   ucm  ::  UseCaseModel
}

exportProduct  ::  Path → InstanceModel → IO ()
exportProduct t  product =do
  exportUcmToLatex (t ∪ " /doc.tex ")  (ucm product)

exportUcmToLatex =...
```

Figura 4.1: Trecho de código da implementação inicial de Hephaestus

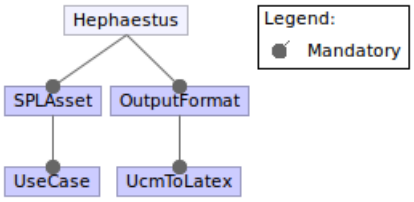

Figura 4.2: Configuração de Hephaestus na primeira versão.

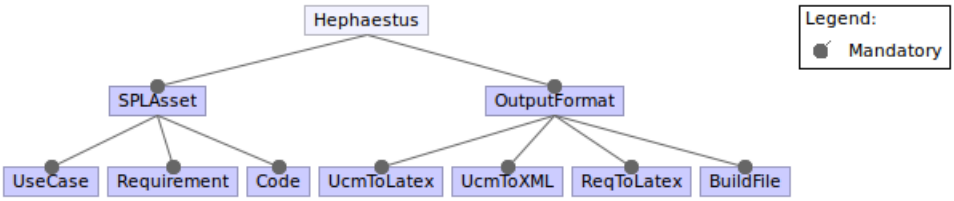

Figura 4.3: Configuração de Hephaestus na segunda versão.

(b) implementar novas funções *parser* e *output* para a leitura/escrita de ativos como requisitos e código-fonte em Hephaestus;

(c) implementar novas transformações para resolver variabilidades em requisitos e código-fonte;

(d) evoluir a função *parser* XML do conhecimento da configuração, para que possa reconhecer a sintaxe concreta das novas transformações;

(e) evoluir a definição da instância do produto base usada pela função `build`;

(f) evoluir os tipos de dados `SPLModel` e `InstanceModel` para envolver os novos ativos.

A evolução de tipos de dados e funções de Hephaestus como descrito acima estão apresentados nas Figuras 4.4 e 4.5. Para introduzir suporte a novos tipos de artefatos, nós temos que alterar ambos os tipos de dados `SPLModel` (linhas 1-6 na Figura 4.4) e `InstanceModel` (linhas 8-15), a definição de produto vazio (linhas 23-25), a função `exportProduct` (linhas 27-31) e a função `xml2Transformation` do parser XML do conhecimento da configuração (Figura 4.5) — mesmo que os tipos de dados `ConfigurationKnowledge` e `FeatureModel` e o interpretador (função `build`) apresentam algum grau de estabilidade pois não precisamos alterar suas implementações quando introduzimos suporte à variabilidade para novos artefatos. Aqui nós poderíamos dizer que os tipos de dados `SPLModel` e `InstanceModel` não são *abertos* desde que para introduzir novos artefatos de LPS nós precisamos alterar essas definições de tipos de dados. Portanto, a arquitetura e a implementação atuais de Hephaestus não são suficientemente flexíveis, pois não somos capazes de introduzir novos tipos de dados representando a sintaxe abstrata de um novo artefato de LPS e suas transformações de forma modular, o que implica em um esforço oneroso para a extensão de Hephaestus.

Em particular, evoluir Hephaestus para suportar variabilidade em código-fonte (Figura 4.4) precisamos inserir um novo tipo de artefato no `SPLModel` (`splComponents` na

linha 5). Este artefato é uma lista de pares que relaciona um nome de arquivo de código-fonte com o seu *path* relativo. O mesmo tipo de artefato foi também introduzido no `InstanceModel` (linha 12). Além disso, dois outros campos foram necessários no `InstanceModel`: (a) `buildEntries` (linha 13), que declara diretivas de pré-processamento; e (b) `preProcessFiles` (linha 14), que declara uma lista de arquivos que devem ser pré-processados. Esses campos são instanciados quando Hephaestus constrói um produto, considerando as transformações apropriadas de uma configuração de produtos.

O trecho de código na Figura 4.5 mostra o impacto na função `xml2Transformation` do parser XML do conhecimento da configuração em Hephaestus. A primeira versão de Hephaestus declara apenas as primeiras quatro sentenças do *case* na função `xml2Transformation` (linhas 8-11). A transformação `selectRequirements` (linha 12) trata a variabilidade nos modelos de requisitos, enquanto que as transformações restantes (linhas 13-16) resolvem variabilidades em código-fonte. Poderíamos dizer que as funções `exportProduct` e `xml2Transformation` não são *abertas*, desde que novos formatos de saída e novos tipos de transformações não podem ser introduzidos de forma modular. Similarmente à falta de tipos de dados *abertos*, essa deficiência reforça a insuficiente flexibilidade da arquitetura e implementação de Hephaestus.

Finalmente, a arquitetura e implementação atuais de Hephaestus também apresentam configurabilidade limitada: para obter uma nova versão de Hephaestus gerenciando variabilidades em apenas um subconjunto apropriado dos artefatos (casos de uso, requisitos, código) atualmente suportados pela ferramenta, por exemplo, uma versão que suporte somente código e casos de uso, o impacto da mudança é similar ao que foi descrito previamente para a adição de novos artefatos em Hephaestus.

4.3 Análise de Domínio de Hephaestus-PL

Embora adaptado para o gerenciamento de variabilidades em uma LPS específica (Ferreira et al., 2010), alguns usuários de Hephaestus poderíam apreciar configurações mais específicas da ferramenta. Por exemplo, alguns usuários poderíam estar interessados em gerenciar a variabilidade somente em requisitos e casos de uso; outros poderíam estar interessados em gerenciar variabilidade somente em código-fonte; e ainda outros engenheiros poderíam estar interessados em gerenciar variabilidades em requisitos, casos de uso e código-fonte. Assim, novas extensões de Hephaestus têm sido recentemente propostas. Por exemplo, atualmente existem variantes de Hephaestus que suportam variabilidade em modelos de processos de negócio (Machado et al., 2011) e artefatos *Simulink* (Steiner et al., 2012).

Estas variantes compartilham os mesmos desafios de configurabilidade e flexibilidade explicados na Seção 4.2. Para endereçar esses desafios, nós adotamos uma perspectiva de LPS ao próprio Hephaestus, tratando assim a comunalidade nessas variantes e sistematicamente gerenciando a variabilidade existente. A partir das versões existentes de Hephaestus, nós adotamos uma estratégia *extrativa* (Krueger, 2001) realizando o *bootstrapping* de Hephaestus-PL a partir de tais variantes. Assim, analisamos as variantes existentes de Hephaestus e manualmente identificamos as features comuns e variáveis, os elementos arquiteturais e de implementação. O restante desta seção explica o resultado desta estratégia para identificar a comunalidade e a variabilidade entre tais elementos. Na Seção 4.4 apresentamos e explicamos como o projeto de domínio de Hephaestus-PL aproveita esta análise de domínio e endereça os requisitos de configurabilidade e flexibi-

```
1  data SPLModel =SPLModel {
2      splFeatureModel :: FeatureModel,
3      splUCM :: UseCaseModel,
4      splReq :: RequirementModel,
5      splComponents :: ComponentModel
6  }
7
8  data InstanceModel =InstanceModel {
9       featureConfiguration :: FeatureConfiguration,
10     ucm :: UseCaseModel,
11     req :: RequirementModel,
12     components :: ComponentModel,
13      buildEntries :: [ PreprocessingDirective ],
14      preProcessFiles :: [ PreprocessingFiles ]
15 }
16
17 build :: FeatureModel → FeatureConfiguration
18         → ConfigurationKnowledge → SPLModel →InstanceModel
19 build fm fc ck spl = derive ts spl emptyInstance
20 where
21      ts =concat [transformations c | c ∈ ck, eval fc (expression c)]
22      ucmodel       =splUCM spl
23      emptyUCM      =ucmodel { useCases =[] , aspects = [] }
24      emptyReq      =RM { reqs =[] }
25      emptyInstance =InstanceModel fc emptyUCM emptyReq [] [] []
26
27 exportProduct :: FilePath → FilePath → InstanceModel → IO ()
28 exportProduct s o product =do
29   exportUcmToLatex (o ∪ "/doc.tex") (ucm product)
30   exportRequirementsToLatex (o ∪ "/doc.lst") (req product)
31   exportSourceCode s o product
32
33 exportSourceCode :: FilePath → FilePath → InstanceModel → IO ()
34 exportSourceCode s o p =do
35   copySourceFiles s o (components p)
36   exportBuildFile (o ∪ "/build.lst") (buildEntries p)
37   preprocessFiles (o ∪ "/build.lst") (preProcessFiles p) o
```

Figura 4.4: Tipos de dados `SPLModel` e `InstanceModel`, definição de `emptyInstance` e função `exportProduct` depois de introduzir suporte ao gerenciamento de variabilidades em requisitos e código-fonte

```
xml2Transformation :: XmlTransformation → Parser  Transformation
xml2Transformation transformation =
let
   args =...
   tnsName =xmlTransformationName transformation
in
   case tnsName of
   '' selectScenarios '' → Success ( selectScenarios  args)
   '' selectUseCases '' → Success (selectUseCases  args)
   '' evaluateAspects '' → Success (evaluateAspects  args)
   '' bindParameter'' → ...
   '' selectRequirements '' → ...
   '' selectComponents'' → ...
   '' selectAndMoveComponent'' →...
   '' createBuildEntries '' → ...
   '' preprocessFiles '' → ...
   otherwise →Fail   ''...''
```

Figura 4.5: Trecho de código usado durante o *parser* XML do conhecimento da configuração

lidade; os detalhes da implementação são apresentados na Seção 4.5. Seção 4.6 detalha o processo reativo necessário para introduzir suporte ao gerenciamento de variabilidades em novos artefatos de forma flexível.

4.3.1 Modelo de Features de Hephaestus-PL

Em termos de espaço do problema, o modelo de features de Hephaestus-PL está representado na Figura 4.7. Como mostra o diagrama, a feature *SPLAsset* é mandatória e a feature *OutputFormat* é opcional e ambas são features pais de *or-features* tal que qualquer combinação de modelos de artefatos e formatos de saída são suportados, desde que atendidas as restrições globais do modelo de features de Hephaestus-PL. Por exemplo, uma determinada instância poderia conter processos de negócio e casos de uso e exportar ambos os artefatos como arquivos XML. Gerenciar variabilidades em tais combinações de artefatos é essencial em Hephaestus-PL e não tem sido endereçado em trabalhos relacionados (Seção 4.8 discute esta questão).

Ainda no diagrama de features da Figura 4.7 aparecem algumas restrições globais que devem ser satisfeitas para toda configuração de features válida de Hephaestus-PL. Para ilustrar, as Figuras 4.2 e 4.3 mostram duas configurações válidas de Hephaestus-PL, enquanto que a Figura 4.6 mostra uma configuração inválida de Hephaestus-PL, em que a feature *UcmToXML* é selecionada, mas a feature *UseCase* não está selecionada. Neste caso, a restrição $UcmToXML \vee UcmToLatex \Rightarrow UseCase$ foi violada levando a uma configuração de feature inválida de Hephaestus-PL.

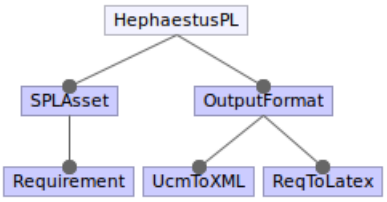

Figura 4.6: Configuração inválida de Hephaestus-PL

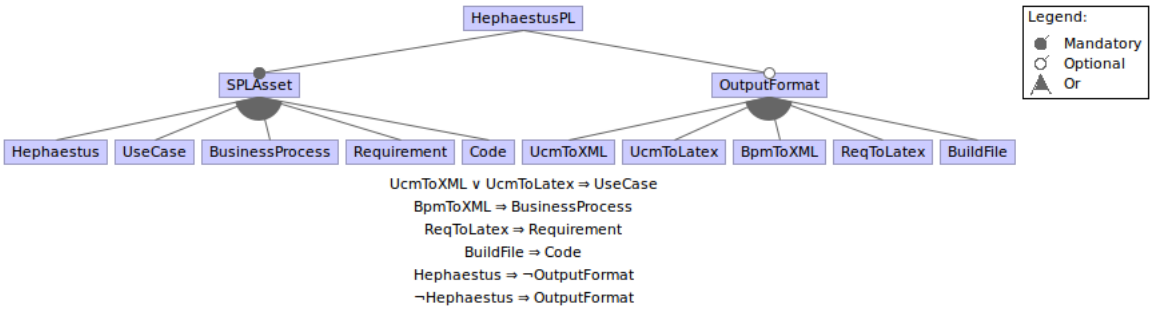

Figura 4.7: Modelo de Features de Hephaestus-PL. Nesta versão, as features *SPLAsset* e *OutputFormat* definem um relacionamento *or-feature* com suas features filhas

4.3.2 Comunalidade e Variabilidade de Hephaestus-PL

No espaço da solução, existe uma quantidade significante de comunalidades entre essas versões: a função `build`, bem como outras funções e tipos de dados, são compartilhados entre todas essas variantes de Hephaestus. Por outro lado, variabilidade tem forma regular necessitando ambos tipos de dados e funções *abertas*, como explicado na Seção 4.2.1. Em particular, o resultado da análise de domínio está consistente com a história da evolução relatada anteriormente e revela que a comunalidade reside no seguinte:

- representação da configuração do produto (FC): um tipo de dado algébrico representa uma configuração de features válida;
- representação do modelo de features (FM): um tipo de dado algébrico representa o modelo de features de uma linha de produtos;
- representação básica do conhecimento da configuração (CK): um tipo de dado algébrico representa o `ConfigurationKnowledge` da linha de produtos;
- instanciação do produto: a função `build` executa a instanciação da LPS pela geração de um produto correspondente à configuração específica da linha de produtos.

Por outro lado, a análise de domínio revelou que a variabilidade de Hephaestus-PL reside no seguinte:

- **Representação do Ativo**: tipos de dados algébricos que representam a sintaxe abstrata de diferentes ativos de LPS, tais como casos de uso, processos de negócio e código-fonte.

- **Transformações do Ativo**: funções que manipulam tais ativos resolvendo as variabilidades dos ativos da LPS. Algumas transformações basicamente selecionam um ativo específico da linha de produtos incluíndo-o no produto durante o processo de derivação de produtos. Outras transformações mudam a estrutura de um ativo da LPS no produto final.
- **Entrada/Saída do Ativo**: funções de *parser/output* para a leitura/escrita de ativos convertendo a sintaxe concreta de um ativo para a sintaxe abstrata correspondente em Hephaestus-PL, ou seja, os tipos de dados abstratos do ativo.
- **Empacotamento do Ativo**: tipos de dados algébricos `SPLModel` e `InstanceModel` que incluem o conjunto de ativos de LPS e o campo modelo de features (ou campo configuração de features, no caso do `InstanceModel`) de uma dada instância de Hephaestus-PL.
- **Instância Vazia**: define a representação inicial de um produto durante a atividade de derivação de produtos. É uma instância do tipo de dados `InstanceModel` e serve como uma *baseline* que é sucessivamente refinada pela função `build` (ver Figura 4.4) até que todas as transformações apropriadas tenham sido aplicadas e o produto final derivado.
- **Parser do CK** (função `xml2Transformation`): realiza o reconhecimento da sintaxe concreta das transformações para popular o CK com os ativos de LPS.

Embora, no nível de domínio uma configuração específica de Hephaestus representada pela combinação de *or-features* seja conceitualmente simples, no espaço da solução isso é bem mais complexo porque representa gerenciar a variabilidade em uma combinação de ativos em diferentes níveis de granularidade, ou seja, granularidade grossa e fina, e a implementação das features preserva um certo grau de espalhamento e entrelaçamento no código-fonte de Hephaestus. Por exemplo, para introduzir suporte à variabilidade para um novo ativo de LPS em Hephaestus, é necessário implementar os tipos de dados para representar a sintaxe abstrata desses modelos, implementar as transformações para resolver a variabilidade, implementar as funções de *parser/output* para a leitura/escrita desses ativos em Hephaestus, e ainda, estender alguns tipos de dados e funções de Hephaestus conforme detalhado na Seção 4.2.1.

4.4 Projeto de Domínio de Hephaestus-PL

Esta seção introduz o projeto de Hephaestus-PL apresentando uma visão estática quando descrevemos a sua arquitetura na Subseção 4.4.1 e uma visão dinâmica quando descrevemos o processo de derivação de novas instâncias de Hephaestus-PL na Subseção 4.4.2. Além disso, descrevemos os principais elementos da arquitetura de Hephaestus-PL na Subseção 4.4.3.

A arquitetura de Hephaestus-PL está representada através de um diagrama de pacotes da UML com algumas relações de dependência e onde os pacotes apresentados correspondem aos principais blocos dessa arquitetura.

4.4.1 Arquitetura de Hephaestus-PL

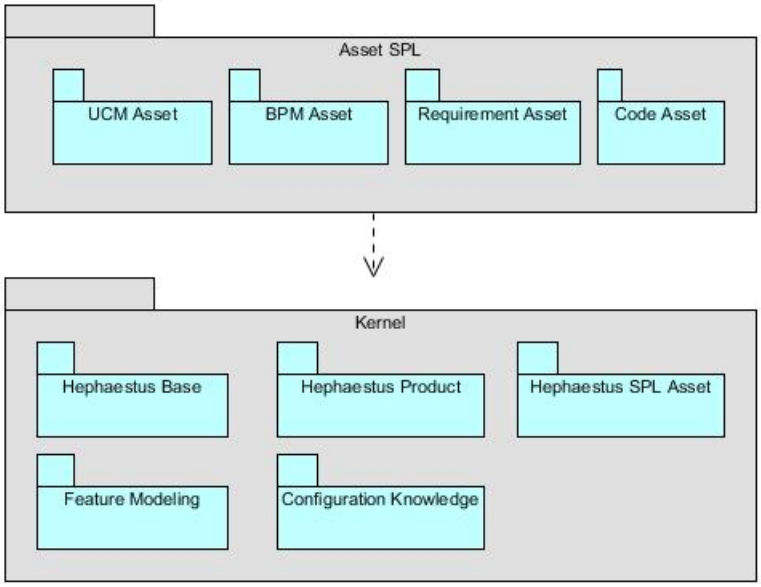

Figura 4.8: Arquitetura de Hephaestus-PL

A fim de alcançar os requisitos de configurabilidade e flexibilidade em ferramentas de derivação de produtos, tais como Hephaestus, propomos o desenvolvimento dessas ferramentas como próprias linhas de produtos. Neste caso, Hephaestus-PL como uma linha de produtos das variantes existentes de Hephaestus. E, para o gerenciamento da variabilidade dessas linhas de produtos utilizamos uma abordagem transformacional escolhida a partir dos resultados obtidos com o estudo comparativo de mecanismos de desenvolvimento de linhas de produtos apresentado no Capítulo 3. Para atender ao modelo de domínio identificado na Seção 4.3 e, assim, gerenciar a variabilidade, o projeto de Hephaestus-PL compreende a arquitetura ilustrada na Figura 4.8. O componente chave nesta arquitetura é o *Kernel* representando a estrutura mínima necessária para a geração de instâncias de produtos de Hephaestus-PL além da possibilidade de gerar-se a si mesmo em um processo de *bootstrapping*. Dentro do *kernel*, o pacote *Hephaestus Base* representa a comunalidade entre todas as instâncias de Hephaestus-PL e apresenta pontos de variabilidade que são resolvidos por transformações definidas no pacote *Hephaestus SPL Asset* em um processo de derivação coordenado pelo pacote *Hephaestus Product*. Além disso, o *Kernel* contém ainda a definição da representação do Modelo de Features e do Conhecimento da Configuração que são elementos chaves de uma instância de Hephaestus-PL no processo de derivação de produtos.

Além do *Kernel*, a arquitetura de Hephaestus-PL apresenta o *SPL Assets* que contribui com o *Kernel* na geração de instâncias de Hephaestus-PL. *SPL Assets* representa a definição de cada ativo e suas transformações correspondentes que permitem a geração de instâncias de Hephaestus-PL que suportem o gerenciamento de variabilidade e a derivação de produtos no domínio desses ativos. Na Subseção 4.4.3 apresentamos os detalhes dos elementos descritos na Figura 4.8.

Adotamos uma abordagem transformacional (Schaefer et al., 2010b) para o gerenciamento da variabilidade em Hephaestus-PL devido à expressividade necessária para endereçar a heterogeneidade de variabilidades observadas na Seção 4.3.2, sem comprometer a modularidade e a compreensibilidade e, conseqüentemente, a flexibilidade que são problemas na abordagem anotativa (Kästner et al., 2008). Embora essas variabilidades envolvem espalhamento e entrelaçamento de features, uma abordagem composicional como, por exemplo, POA não atenderia à expressividade necessária dada a heterogeneidade na granularidade das variabilidades. Com a abordagem transformacional proposta aqui observamos que a *i)* configurabilidade está endereçada pois habilita Hephaestus-PL com um suporte automático para gerar instâncias de produtos com diferentes combinações de ativos; e a *ii)* flexibilidade é garantida pelo projeto arquitetural de Hephaestus-PL que oferece suporte ao gerenciamento de variabilidades em diferentes ativos, independência entre os ativos e, principalmente, nenhum impacto no *Kernel* ao inserir novos ativos em Hephaestus-PL.

4.4.2 Processo de Derivação de Produtos em Hephaestus-PL

Para ilustrar um cenário de processo de derivação de produtos em Hephaestus-PL, suponha uma configuração de features (FC) específica contendo quatro features: modelo de caso de uso (*UseCase*), modelo de processo de negócio (*BusinessProcess*), caso de uso no formato XML (*UcmToXML*) e processo de negócio no formato XML (*BpmToXML*), como mostrado na Figura 4.9.

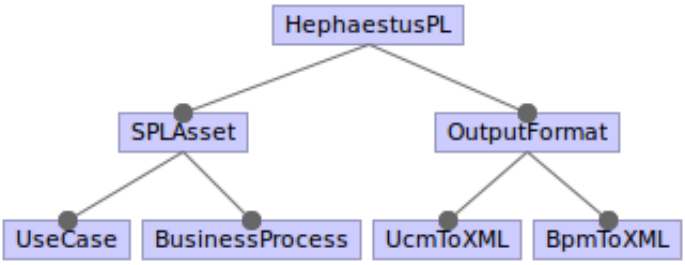

Figura 4.9: Configuração de features para geração de uma instância de Hephaestus-PL

O processo de derivação de produtos em Hephaestus-PL é baseado no conhecimento da configuração (CK) de Hephaestus-PL, um trecho do qual está apresentado na Tabela 4.1. Esse CK relaciona expressões de features com as transformações que resolvem a variabilidade de Hephaestus-PL. Como mostrado na coluna *Transformações* da Tabela 4.1, definimos transformações de Hephaestus-PL que estão detalhadas na Subseção 4.4.3, e quando aplicadas, elas progressivamente estendem construções sintáticas do produto Hephaestus base diminuindo a variabilidade relacionada aos ativos selecionados na configuração conforme Figura 4.10, até a geração da instância final de Hephaestus-PL.

Figura 4.10 descreve de forma abstrata os passos realizados no processo de derivação de produtos de Hephaestus-PL considerando a configuração de features mostrada na Figura 4.9. O CK de Hephaestus-PL (Tabela 4.1) guia esse processo transformacional para a geração de uma instância de Hephaestus-PL a partir do produto Hephaestus base

Expressões de Features	**Transformações**
True	SelectHephaestusBase
UseCase	SelectAsset ”Use Case”
UseCase AND UcmToXML	SelectExport ”UcmToXML”
UseCase AND UcmToLatex	SelectExport ”UcmToLatex”
BusinessProcess	SelectAsset ”Business Process”
BusinessProcess AND BpmToXML	SelectExport ”BpmToXML”
Requirement	SelectAsset ”Requirement”
Requirement AND ReqToLatex	SelectExport ”ReqToLatex”
Code	SelectAsset ”Code”
Code AND BuildFile	SelectExport ”BuildFile”

Tabela 4.1: Trecho do Conhecimento da Configuração de Hephaestus-PL

(nesse caso, em cinco passos). O CK de Hephaestus-PL é avaliado da primeira até a última linha, desencadeando a execução daquelas transformações cuja expressão de features correspondente for avaliada como verdadeira de acordo com a configuração de features dada.

Assim, o processo de derivação de produtos de Hephaestus-PL começa com a execução da transformação `SelectHephaestusBase` associada com a expressão de features `True` na primeira linha do CK de Hephaestus-PL (Tabela 4.1). Com efeito, esta transformação é sempre executada no início da geração de uma nova instância de Hephaestus-PL e representa a seleção do produto Hephaestus base que contém a comunalidade de uma instância de Hephaestus-PL.

Em seguida, na segunda linha do CK de Hephaestus-PL, a expressão de features `UseCase` é avaliada como verdadeira de acordo com o FC mostrado na Figura 4.9 e, portanto, a transformação `SelectAsset "UseCase"` é executada chamando um conjunto de transformações de baixo nível no produto Hephaestus base, ligando pontos de variação do Hephaestus base através da adição de componentes do *UCM Asset* (passo 2 na Figura 4.10). Mais especificamente, a transformação introduz o tipo de dados do artefato UCM como campo nos tipos de dados `SPLModel` e `InstanceModel` e importa os módulos que definem os tipos de dados UCM e as transformações correspondentes. Em seguida no passo 3, a expressão de features `UseCase AND UcmToXML` é avaliada como verdadeira na terceira linha do CK de Hephaestus-PL e, então, a transformação `SelectExport "UcmToXML"` é executada. Esta transformação estende o produto em derivação, adicionando componentes do *UCM Asset* para suportar a exportação de casos de uso no formato XML, ou seja, introduz o construtor `ExportUcmXML` correspondente à feature *UcmToXML* dentro do tipo de dados `ExportModel` e adiciona o módulo que implementa o formato de saída XML para casos de uso no *Product'*.

Então, os passos 4 e 5 executam, no produto em derivação, transformações semelhantes àquelas realizadas pelas passos 2 e 3 descritos anteriormente. Portanto, as expressões de features `BusinessProcess` e `BusinessProcess AND BpmToXML` são avaliadas como verdadeiras de acordo com a configuração de features de entrada. Como resultado, as transformações correspondentes no CK resolvem mais alguns pontos de variação restantes do produto em derivação, adicionando componentes do *BPM Asset* neste produto

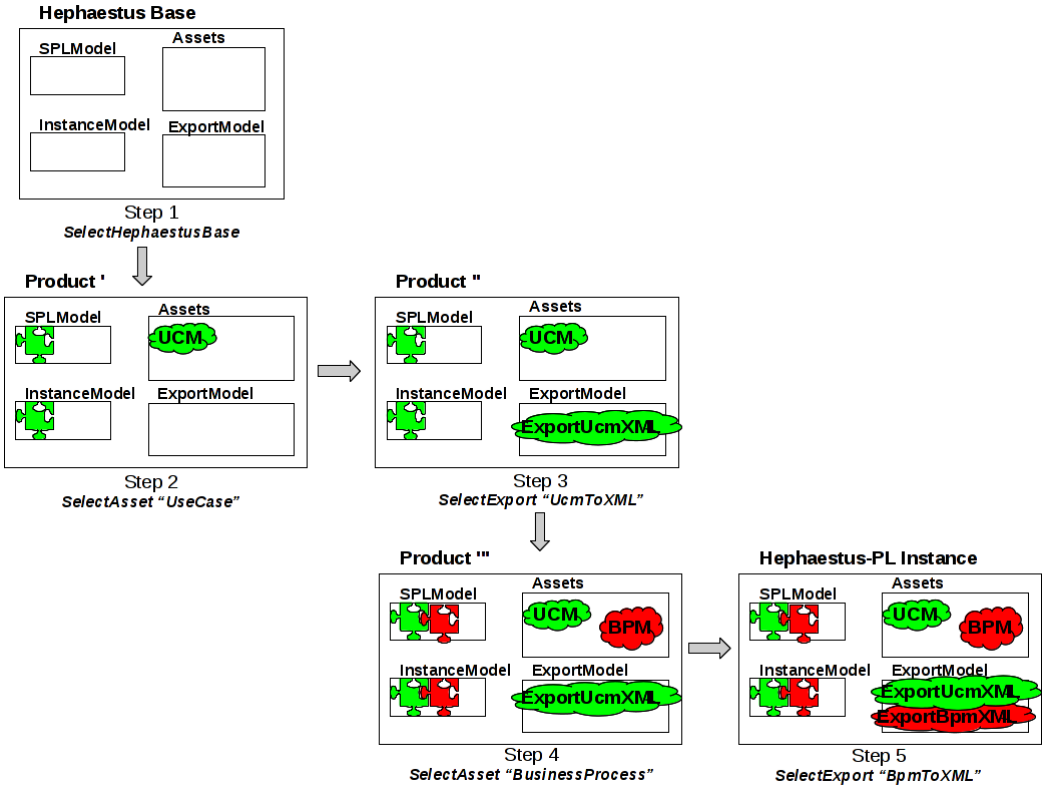

Figura 4.10: Derivação de uma instância de Hephaestus-PL que suporta as features UCM, BPM, UcmToXML e BpmToXML.

para suportar o gerenciamento de variabilidades e a capacidade de exportação de modelo de processos de negócios na instância gerada de Hephaestus-PL.

Observamos que ambos *UCM Asset* and *BPM Asset* são elementos arquiteturais pertencentes ao *SPL Assets*, como representado na Figura 4.8.

4.4.3 Elementos Arquiteturais de Hephaestus-PL

A seguir, detalhamos os principais componentes da arquitetura de Hephaestus-PL com ênfase aos pacotes do *Kernel* e do *SPL Assets*. O *Kernel* consiste de cinco pacotes: *Produto Hephaestus*, *Hephaestus Base*, *Asset Hephaestus SPL* e as representações do *Modelo de Features* e *Conhecimento da Configuração*. Juntos esses elementos compreendem a estrutura mínima responsável pela geração de instâncias de produtos Hephaestus-PL. Além disso, eles têm um certo grau de acoplamento e são principalmente estáveis em relação à evolução de Hephaestus-PL. Descrevemos ainda nesta Seção 4.4.3 o pacote arquitetural *SPL Assets*.

Produto Hephaestus

Produto Hephaestus é uma instância mínima de Hephaestus-PL que suporta somente o gerenciamento de variabilidade do *Asset Hephaestus SPL* e corresponde a um módulo Haskell que pode ser gerado por *bootstrapping*, isto é, *Produto Hephaestus* pode ser usado

```
1  build :: FeatureModel
2        → FeatureConfiguration
3        → ConfigurationKnowledge
4        → SPLModel
5        → InstanceModel
6
7  build fm fc ck spl =stepRefinement ts spl emptyInstance
8    where emptyInstance =mkEmptyInstance fc spl
9          ts =tasks ck fc
10
11 tasks :: ConfigurationKnowledge → FeatureConfiguration → [TransformationModel]
12 tasks ck fc =concat [transformations c | c ∈ ck, eval fc (expression c)]
13
14 transform :: TransformationModel →SPLModel →InstanceModel →InstanceModel
15 transform (HephaestusTransformation t) s i =transformHpl t s i
```

Figura 4.11: Trecho de código do *Produto Hephaestus*

para gerar-se a si próprio, caso a feature *Hephaestus* seja selecionada numa configuração de FM de Hephaestus-PL.

Gerenciar a variabilidade do *Asset Hephaestus SPL* significa que o *Produto Hephaestus* é usado para derivar produtos de acordo com o espaço de variabilidade de Hephaestus-PL. Na verdade, este módulo declara a função `build` que controla o processo de derivação de produtos de Hephaestus-PL – por refinamentos progressivos do *Hephaestus Base* – e fornece uma definição simples para a função `transform` (trecho de código na Figura 4.11). Nesta instância particular do *Produto Hephaestus*, a função `transform` somente trata com as transformações do *Asset Hephaestus SPL*, que estão detalhadas no subtópico *Asset Hephaestus SPL* desta Seção 4.4.3.

Hephaestus Base

O *Hephaestus Base* é um módulo Haskell que serve como um módulo base para a derivação de instâncias Hephaestus-PL. Ele representa a comunalidade entre as instâncias de Hephaestus-PL e foi obtido a partir da análise de domínio de Hephaestus-PL (Seção 4.3.2). Quando construímos uma instância de Hephaestus-PL, o processo de derivação primeiro lê este módulo e, em seguida, executa transformações do asset Hephaestus (principalmente `SelectAsset` e `SelectExport`) para atender as features selecionadas, refinando várias definições do módulo `HephaestusBase.hs`. Figura 4.12 apresenta um trecho de código do módulo do *Hephaestus Base* cujas definições e declarações são listadas a seguir:

- representações básicas para os tipos de dados `SPLModel` (linhas 3-5) e `InstanceModel` (linhas 7-9), onde o primeiro tem apenas o campo de dados `featureModel` e o tipo de dados `InstanceModel` tem apenas o campo de dados `featureConfiguration`;
- o tipo de dados `TransformationModel` (linha 11) que suporta apenas o construtor `UndefinedTransformation`;

- um caso para a função `transform` (linhas 13-14), que suporta apenas a transformação `UndefinedTransformation`;
- o tipo de dados `ExportModel` (linha 16) que suporta apenas o construtor `UndefinedExport`;
- uma lista vazia (`lstExport`) (linhas 18-19) de tipos de dados de exportação do asset usados para exportar um produto em diferentes formatos de saída;
- um caso para a função `export` (linhas 21-22), que também suporta apenas o formato de exportação `UndefinedExport`;
- uma função `mkEmptyInstance` (linhas 24-27) que retorna um `InstanceModel` contendo somente o campo de dados `featureConfiguration`;
- uma função `xml2Transformation` (linhas 29-30) que compreende o processo de *parser* XML do CK para o reconhecimento da sintaxe concreta das transformações do asset da instânca de Hephaestus-PL; e
- uma função `main` (linhas 32-39) para executar a instância de Hephaestus-PL e a geração de produtos dos ativos gerenciados na instância de Hephaestus-PL.

Os tipos de dados `TransformationModel` e `ExportModel` podem ser interpretados como pontos de variação que serão resolvidos através da introdução de novos valores que substituirão as definições `UndefinedTransformation` e `UndefinedExport`, respectivamente, no refinamento da instância de Hephaestus-PL.

Além disso, instâncias de Hephaestus-PL devem fornecer uma definição para a função `transform` para cada asset LPS suportado onde o processo de derivação de produtos de Hephaestus-PL refina a função `transform` introduzindo essas novas definições através de metaprogramação em Haskell.

A linha 36 da Figura 4.12 representa uma instância `undefined` gerada de Hephaestus-PL (variável `product`). Esta linha será substituída no produto emergente pela linha `let product = build fm fc cm spl` que executa a função `build` para a geração de uma instância concreta de produto Hephaestus-PL a partir de uma configuração de features.

Asset Hephaestus SPL

Conforme apresentado na Figura 4.14, este elemento da arquitetura de Hephaestus-PL declara o tipo algébrico de dado que representa o *Asset Hephaestus SPL*, que corresponde a uma lista de módulos Haskell (linha 1), e o conjunto de transformações (linhas 3-8) para o gerenciamento da variabilidade de Hephaestus-PL. Na essência, essas transformações são responsáveis pelo refinamento do produto Hephaestus base durante a derivação de instâncias Hephaestus-PL. Além disso, este módulo também declara a função `transformHpl` (linhas 10-13) que é responsável por executar as correspondentes transformações de Hephaestus-PL e tem a mesma assinatura da função `transform` declarada no *Produto Hephaestus* descrito anteriormente.

Considerando os requisitos de configurabilidade e flexibilidade guiando o projeto de Hephaestus-PL, nós empregamos um *design* em camadas conforme ilustrado na Figura 4.13. Nós definimos no *Asset Hephaestus SPL* dois conjuntos de transformações: transformações Hephaestus-PL (API de alto nível) e operações de metaprogramação (API de baixo nível).

```
1  module HephaestusBase where
2
3  data SPLModel =SPLModel {
4    featureModel :: FeatureModel
5  }
6
7  data InstanceModel =InstanceModel {
8     featureConfiguration :: FeatureConfiguration
9  } deriving (Data, Typeable)
10
11 data TransformationModel =UndefinedTransformation
12
13 transform :: TransformationModel →SPLModel →InstanceModel →InstanceModel
14 transform UndefinedTransformation _ _ =undefined
15
16 data ExportModel =UndefinedExport
17
18 lstExport ::[ ExportModel]
19 lstExport = []
20
21 export :: ExportModel →FilePath → InstanceModel → IO()
22 export UndefinedExport _ _ =undefined
23
24 mkEmptyInstance :: FeatureConfiguration → SPLModel →InstanceModel
25 mkEmptyInstance fc spl =InstanceModel {
26    featureConfiguration =fc
27 }
28
29 xml2Transformation :: String → [String] → ParserResult TransformationModel
30 xml2Transformation "Undefined" _ =undefined
31
32 main :: IO ()
33 main =do
34  ...
35    let spl =SPLModel { featureModel =fm }
36    let product =undefined
37    let out =(outputFile (snd t) (snd n))
38    sequence_ [export x out product | x ∈ lstExport]
39  ...
```

Figura 4.12: Trecho de código do módulo *Hephaestus Base* de Hephaestus-PL

Cada transformação Hephaestus-PL é implementada chamando os serviços de operações de metaprogramação. O *design* de cada camada foi orientado pela análise de domínio em termos de assets de implementação: a API de alto nível foi guiada pelo resultado da análise de domínio de Hephaestus-PL (Seção 4.3.2), enquanto que a API de baixo nível, pela análise de domínio das APIs de alto nível. Mais detalhes sobre as APIs de baixo nível são apresentados na Seção 4.5.3.

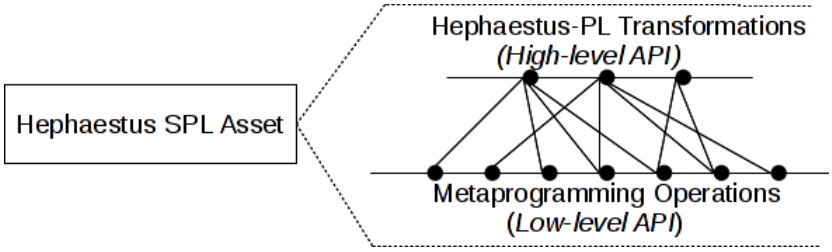

Figura 4.13: Visão Lógica das APIs de Hephaestus-PL

Definimos seis transformações Hephaestus-PL que são representadas pelos seguintes construtores:

- `SelectHephaestusBase` é sempre a primeira transformação a ser executada no processo de derivação de uma instância de Hephaestus-PL. Esta transformação seleciona um produto Hephaestus base que representa a comunalidade de um produto Hephaestus-PL.
- `SelectAsset` refina o produto a ser derivado dando suporte à variabilidade dos assets selecionados como descrito pela configuração de features. Ou seja, essa transformação estende os tipos de dados `SPLModel`, `InstanceModel` e `TransformationModel` e as funções `transform`, `xml2Transformation` e `mkEmptyInstance` que representa a definição de instância vazia para a função `build` manipular tais assets. Além disso, incorpora os módulos que definem os tipos de dados algébricos, as transformações e o *parser* do asset selecionado no produto sendo gerado. Também executa extensões na função `main` introduzindo a instrução *parser* do asset e estendendo a instância do tipo de dados `SPLModel` com o asset selecionado para entrada do processo `build`.
- `SelectExport` refina o produto em derivação introduzindo suporte ao formato de saída selecionado estendendo o tipo de dados `ExportModel`, a definição da lista `lstExport` e a função `export`. Além disso, incorpora o módulo que implementa a função do formato de saída selecionada no produto sendo gerado. Definimos diferentes transformações como `SelectAsset` e `SelectExport` para representar o refinamento sobre o produto Hephaestus base porque o conjunto de operações de metaprogramação associadas com cada uma dessas transformações Hephaestus-PL é diferente e independente.
- `BindProductName` renomeia o módulo de uma instância Hephaestus-PL.

```
1  data HephaestusModel =HephaestusModel [HsModule]
2
3  data HephaestusTransformation =SelectHephaestusBase
4                                | SelectAsset  String
5                                | SelectExport  String
6                                | BindProductName String
7                                | RemoveProductMainFunction
8                                | SelectCKParser
9
10 transformHpl  ::  HephaestusTransformation
11               → SPLModel
12               → InstanceModel
13               → InstanceModel
```

Figura 4.14: Trecho de código do *Asset Hephaestus SPL*

- `RemoveProductMainFunction` é somente utilizada quando a feature *Hephaestus* está em uma configuração de FM de Hephaestus-PL. Essa transformação remove a definição da função `main` do produto sendo derivado porque a feature *Hephaestus* não a utiliza, utilizando em seu lugar a função `buildHpl` que está localizada em outro módulo chamado *IO.hs*.

- `SelectCKParser` refina o produto sendo derivado com o *parser* do CK introduzindo sentenças na função `main` para executar o parsing do CK da instância de Hephaestus-PL e alterando a declaração `product` de `undefined` para `build fm fc cm spl`, que refere-se ao CK definido pela transformação `SelectCKParser`. Além disso, incorpora o módulo que define o parser XML do CK no produto emergente. Essa transformação é aplicada somente quando a expressão de features `NOT Hephaestus` do CK de Hephaestus-PL for avaliada como verdadeira. Quando a feature `Hephaestus` é selecionada, essa transformação não é aplicada porque a instância correspondente a essa feature é o produto Hephaestus e as transformações para o gerenciamento da variabilidade de Hephaestus são precisamente as transformações descritas nesta lista de itens e que são referenciadas a partir do produto Hephaestus e, portanto, não precisam sofrer *parsing*.

Modelo de Features e Conhecimento da Configuração

Estes são módulos Haskell que declaram tipos de dados algébricos para representar os ativos modelo de features (FM) e conhecimento da configuração (CK), bem como funções para verificação de tipos e outros tipos de verificação para esses ativos. Na verdade, estes módulos foram completamente reutilizados das versões anteriores de Hephaestus. Por esta razão, instâncias de Hephaestus-PL compartilham esses ativos como eles são declarados no kernel de Hephaestus-PL— nenhuma transformação sobre esses ativos são necessárias.

As instâncias do FM e CK de Hephaestus-PL são entradas chaves no processo `build` do *Produto Hephaestus* para derivar novas instâncias de Hephaestus-PL.

O FM de Hephaestus-PL declara o espaço da variabilidade de Hephaestus-PL, como mostrado na Figura 4.7. O CK completo de Hephaestus-PL está representado pela Ta-

bela 4.1 adicionando-se as linhas relacionadas à feature `Hephaestus` apresentadas na Tabela 4.2.

O processo de *bootstrapping* de Hephaestus-PL representa a geração de uma instância de Hephaestus-PL usando uma configuração de features que tem apenas a feature `Hephaestus` selecionada. No processo de derivação de produtos Hephaestus-PL esta configuração de entrada vai desencadear a geração de um produto Hephaestus-PL que gerencia a variabilidade do asset Hephaestus. Diferentemente, se usarmos uma configuração de features que tem ambas as features `UseCase` e `BusinessProcess`, a derivação de produtos gerará uma instância Hephaestus-PL que gerenciará variabilidades relacionadas a esses assets selecionados.

Expressão de Features	**Transformações**
Hephaestus	BindProductName "Hephaestus" SelectAsset "Hephaestus" RemoveProductMainFunction
NOT Hephaestus	SelectCKParser

Tabela 4.2: Linhas do CK de Hephaestus-PL relacionadas à feature `Hephaestus`

SPL Assets

Este pacote define os módulos Haskell para cada ativo LPS, ou seja, os tipos de dados algébricos que representam um ativo LPS, o conjunto de transformações para resolver a variabilidade do ativo e a função *parser* para converter o formato externo para os tipos de dados algébricos definidos em Hephaestus-PL.

Além disso, o módulo que define as transformações do ativo LPS também precisa definir um tipo de dados e uma função que compreendam todas as transformações do ativo para serem utilizados na função `transform` durante o refinamento do produto Hephaestus base quando o ativo é selecionado, por exemplo, o tipo de dados `UseCaseTransformation` e a função `transformUcm` para o ativo `UseCase`. A função `transformUcm` deve ter a mesma assinatura da função `transform` declarada no produto Hephaestus base descrito na Seção 4.4.3.

4.5 Implementação de Hephaestus-PL

Nesta seção apresentamos os detalhes da implementação da arquitetura de Hephaestus-PL mostrada na Figura 4.8 em termos de módulos e suas dependências, conforme ilustrado na Figura 4.15.

A linha pontilhada compreende os módulos que compõem cada um dos elementos principais na arquitetura de Hephaestus-PL, por exemplo, os módulos `HephaestusBase.hs` and `HephaestusBaseTypes.hs` são o *Hephaestus Base* do Kernel de Hephaestus-PL. Nós apresentamos os módulos do *nível de domínio* que suportam a geração de novas instâncias de Hephaestus-PL e os módulos do *nível de instância* que foram desenvolvidos para testar Hephaestus-PL com uma configuração de features de entrada. Além disso, apresentamos as operações de metaprogramação que implementam as transformações em código Haskell

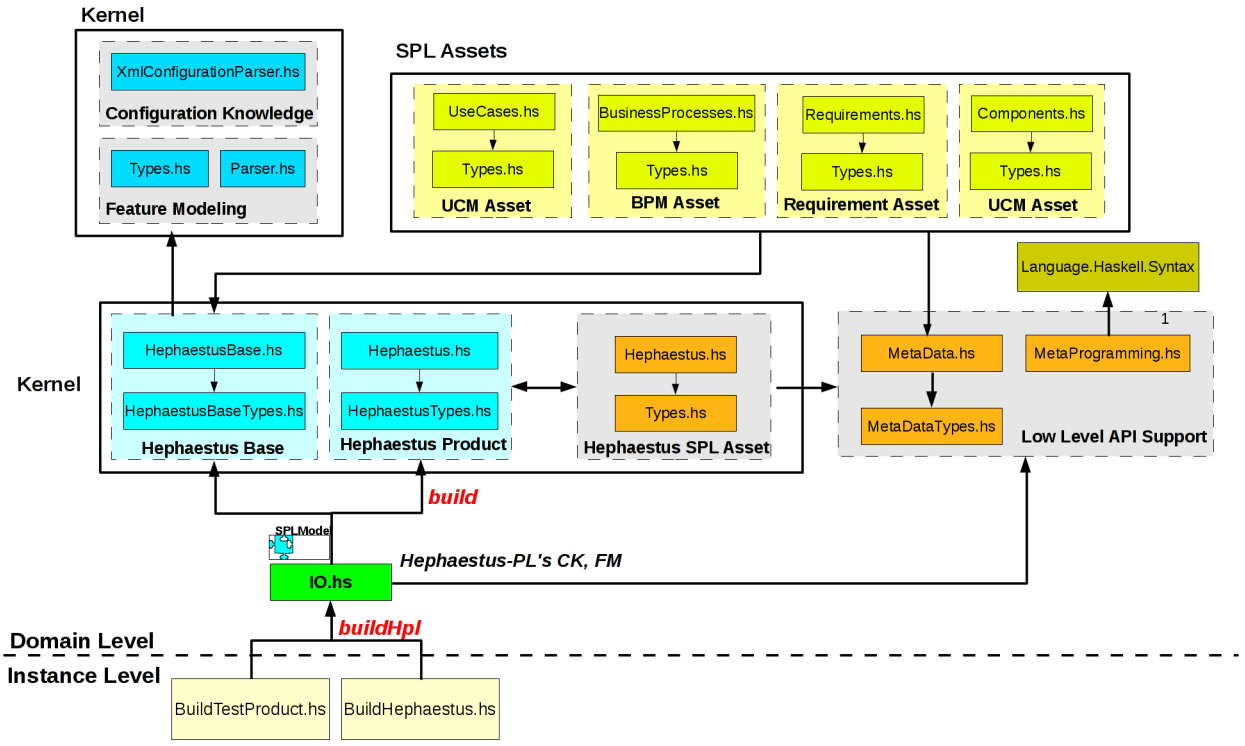

Figura 4.15: Dependência de módulos de Hephaestus-PL

de um *Hephaestus Base* da instância de Hephaestus-PL. O código-fonte de Hephaestus-PL está publicamente[2] disponível.

4.5.1 Módulos e suas Dependências

No nível de domínio, o módulo `IO.hs` contém a função `buildHpl` que representa a interface com o nível de instância de Hephaestus-PL. Esta função é o ponto inicial para a geração de uma nova instância de Hephaestus-PL pelo processo de derivação de produtos de Hephaestus-PL. A função `buildHpl` recebe uma configuração de features do FM de Hephaestus-PL como entrada e prepara o ambiente para executar a geração de uma nova instância de Hephaestus-PL através de um processo guiado pela função `build` do *Produto Hephaestus*. As entradas para o processo de derivação de Hephaestus-PL são quatro: uma instância do tipo de dados `SPLModel`, uma configuração de features válida e os FM e CK de Hephaestus-PL. Neste contexto, a instância do tipo de dados `SPLModel` empacota o ativo Hephaestus, ou seja, os módulos físicos do produto *Hephaestus Base* de Hephaestus-PL são inseridos dentro de uma instância do tipo de dados `HephastusModel` que representa o tipo abstrato do ativo Hephaestus que, por sua vez, está empacotado no tipo de dados `SPLModel`. Além disso, o processo de derivação de Hephaestus-PL precisa do modelo de features (FM) e do conhecimento de configuração (CK) de Hephaestus-PL para executar a função `build` do *Produto Hephaestus* que controla a geração de uma nova instância de Hephaestus-PL. O FM e CK de Hephaestus-PL estão contidos no módulo

[2]https://gitorious.org/hephaestus-pl/hephaestus-pl

`MetaData.hs`. Em seguida, a função `build` contida no módulo `Hephaestus.hs` do *Produto Hephaestus* é executada e retorna uma nova instância da ferramenta Hephaestus-PL. Portanto, o módulo `IO.hs` depende dos módulos do *Hephaestus Base*, *Produto Hephaestus* e `MetaData.hs` como mostra a Figura 4.15.

Os módulos do *Produto Hephaestus* dependem fortemente dos módulos do *Asset Hephaestus SPL* que implementam os tipos de dados `HephaestusModel` e `HephaestusTransformation` e a função `transformHpl`, ou seja, a função `build` avalia o CK e chama a função `transform` definida no mesmo módulo `Hephaestus.hs` em *Produto Hephaestus* que executa a função `transformHpl` no módulo `Hephaestus.hs` do *Asset Hephaestus SPL* para resolver a transformação Hephaestus-PL.

Os módulos referentes ao FM e CK são importados pelos módulos do *Hephaestus Base*.

Os módulos `MetaData.hs`, `MetaDataTypes.hs` e `MetaProgramming.hs` associados ao *Asset HephaestusSPL* compõem o grupo chamado *Low Level API Support* que implementam as operações de metaprogramação para resolver os pontos de variação nos módulos do *Hephaestus Base*. O módulo `MetaProgramming.hs` depende do pacote Language.Haskell.Syntax.

O bloco *SPL Assets* é formado por pares de módulos: o módulo `Types.hs` que define os tipos de dados algébricos do ativo para inserir nos tipos de dados `SPLModel` e `InstanceModel` da instância de Hephaestus-PL e também define os tipos de dados que serão utilizados como construtores nos tipos de dados `TransformationModel` e `ExportModel` do módulo `HephaestusBaseTypes.hs`. Além disso, outro módulo (`nomeAtivo.hs`) de cada par de módulos do bloco *SPL Assets* define as funções que serão usadas nas funções `transform`, `mkEmptyInstance` e `export` do módulo `HephaestusBase.hs`. Assim, os módulos do *SPL Assets* dependem dos módulos do *Hephaestus Base*.

4.5.2 Descrição da Abordagem Transformacional

A hipótese base é que certos tipos chaves e funcionalidades de qualquer instância específica de Hephaestus podem ser derivados por metaprogramação. Para isto, consideramos um produto `Hephaestus Base` a partir do qual construímos novos produtos Hephaestus-PL. Todos os produtos gerados devem definir os tipos de dados `SPLModel`, `InstanceModel`, `ConfigurationKnowledge`, `TransformationModel` e `ExportModel`, este último se for selecionada uma feature da *or-feature* `OutputFormat`, uma função de transformação `transform`, uma função de formato de saída `export` e uma função `main` que controla a geração das instâncias de produtos Hephaestus-PL.

Todas as formas de ativos são, portanto, apresentadas de uma forma que eles possam contribuir para as entidades mencionadas cima, tal como apresentado no bloco *SPL Assets* na Figura 4.15. Usamos operadores de metaprogramação que constróem as entidades do produto (no caso, tipos de dados e funções) a partir das partes correspondentes dos módulos do ativo através da adição de partes que incrementam o produto Hephaestus base.

Por causa do objetivo de Hephaestus-PL ser aplicado a si próprio (princípio de *bootstrapping*), a abordagem transformacional mencionada acima está realmente empacotada como um outro tipo de ativo: *Asset Hephaestus SPL*. Isto é, a construção de qualquer instância de Hephaestus é controlada por uma configuração de features relativa ao FM de Hephaestus-PL e ao correspondente CK, tal como apresentado no módulo `MetaData.hs`.

4.5.3 Operações de Metaprogramação

Transformações Hephaestus-PL	Operações de Metaprogramação
SelectAsset assetName	addUpdateCase
	initializeFieldWithFun
	addImportDecl (4x)
	addLetInstruction
	addGeneratorInstruction
	initializeField
	addField (2x)
	addConstructor
	addUpdateCaseList
SelectExport assetFormat	addUpdateCase
	addImportDecl
	addConstructor
	addListElem
SelectHephaestusBase	setModuleName (2x)
	removeImportDecl
BindProductName	setModuleName (2x)
RemoveProductMainFunction	removeFunction
SelectCKParser	addImportDecl
	addGeneratorInstruction
	addLetInstruction (2x)

Tabela 4.3: Mapeamento das transformações Hephaestus-PL com as operações de metaprogramação associadas.

As operações de metaprogramação necessárias para atender as transformações Hephaestus-PL são de diferentes complexidades, como pode ser observado no módulo `MetaProgramming.hs` onde as operações estão implementadas. Além disso, o módulo `MetaData.hs` define a estrutura de metadados que representa o conjunto de dados (`AssetMetaData` e `ExportMetaData`) relacionado ao ativo que dá suporte à execução das operações de metaprogramação, também chamadas de APIs de baixo nível do kernel de Hephaestus-PL. Algumas informações desse conjunto de dados são os identificadores de tipos de dados, campos de dados, funções e módulos do ativo de Hephaestus-PL. Essas informações serão utilizadas pelas operações de metaprogramação para estender os tipos de dados e funções abertas no módulo base de Hephaestus e gerar uma instância de Hephaestus-PL com os ativos selecionados. Inicialmente, descrevemos as operações de metaprogramação mais simples como `setModuleName`, `addImportDecl`, `addField` e `addConstructor`.

A operação `setModuleName` modifica o nome do módulo para que o módulo base de Hephaestus possa ser renomeado em um módulo para o produto emergente, ou seja, a instância de Hephaestus-PL. Esta operação é chamada nas transformações `SelectHephaestusBase` e `BindProductName`.

A operação `addImportDecl` adiciona uma declaração de *import* da linguagem Haskell no módulo principal do produto emergente. Esta operação é necessária para incorporar qualquer ativo adicional na instância de Hephaestus-PL e é chamada nas transformações

```
1 data SPLModel =SPLModel {
2   featureModel :: FeatureModel
3 }
4
5 data InstanceModel =InstanceModel {
6    featureConfiguration :: FeatureConfiguration
7 } deriving (Data, Typeable)
```

Figura 4.16: Tipos de dados `SPLModel` e `InstanceModel` do módulo base de Hephaestus.

```
1 data SPLModel =SPLModel {
2   featureModel :: FeatureModel,
3   splUcm :: UseCaseModel
4 }
5
6 data InstanceModel =InstanceModel {
7    featureConfiguration :: FeatureConfiguration ,
8   ucm :: UseCaseModel
9 } deriving (Data, Typeable)
```

Figura 4.17: Tipos de dados `SPLModel` e `InstanceModel` estendidos para o ativo *UseCase* no módulo da instância de Hephaestus-PL.

(i) `SelectCKParser` para adicionar uma declaração de importação para o *parser* do CK (módulo `CK.Parsers.XML.XmlConfigurationParser.hs`) e *(ii)* `SelectAsset` para adicionar declarações de importação para incorporar os módulos (tipos de dados algébricos, transformações, *parser* e formato de saída) do ativo e o módulo dos tipos de dados do produto emergente (`HephaestusBaseTypes.hs` renomeado).

A operação `addField` estende um tipo determinado de registro com um campo. Esta operação é necessária para a extensão dos tipos de dados `SPLModel` e `InstanceModel` do módulo base de Hephaestus, conforme apresentado na Figura 4.16. Por exemplo, considerando-se a seleção da feature `UseCase` e a correspondente execução da transformação `SelectAsset "UseCase"` a partir do CK de Hephaestus-PL, um conjunto de operações de metaprogramação são executadas, entre elas a operação `addField` com os seguintes parâmetros `addField "InstanceModel" sel` e `addField "SPLModel" sel'` onde *sel* e *sel'* contêm as informações do novo campo para estender os tipos de dados identificados pelo primeiro parâmetro da operação `addField`, no caso os tipos de dados `InstanceModel` e `SPLModel`, respectivamente. Os valores de *sel* e *sel'* são obtidos da estrutura de metadados para o ativo *UseCase*, no caso os metadados *assetSelector* e *assetSelector'* e seus valores correspondem à uma lista de tuplas. Para o ativo *UseCase*, os valores são [("ucm","UseCaseModel")] e [("splUcm","UseCaseModel")], respectivamente. A Figura 4.17 apresenta os tipos de dados `SPLModel` e `InstanceModel` após a execução da operação `addField` aplicada sobre esses tipos de dados, conforme descrito acima.

Existe também a operação `addConstructor` para estender um tipo de dado algébrico com um construtor associado ao ativo além de remover o construtor *default* não utilizado, no caso, o construtor `UndefinedTransformation`. Esta operação é chamada para estender

o tipo de dados `TransformationModel` de Hephaestus para suportar as transformações de diferentes ativos. Observamos que a operação `addConstructor` é limitada pois suporta apenas a adição de um construtor com um único componente e realmente reutiliza esse tipo específico de componente para o nome do construtor. Assim sendo, nós definimos uma operação `addConstructorWithoutArgs` semelhante à `addConstructor` para estender o tipo de dados `ExportModel` de Hephaestus para suportar diferentes formatos de saída de ativos. Neste caso, a operação `addConstructorWithoutArgs` apenas suporta a adição de um construtor sem parâmetros. Nós também removemos a declaração de ponto de variação `UndefinedExport` nesta operação.

A operação `addListElem` acrescenta na lista `lstExport` os construtores do tipo de dados `ExportModel`.

A adição de campos e construtores é relativamente simples no nível de tipos, mas precisamos também de operações não tão triviais que transformem funções que prontamente usam os tipos de dados afetados pela seleção de ativos na instância de Hephaestus-PL. Existem as operações `initializeField` e `initializeFieldWithFun` que modificam todas as expressões para a construção de um registro para um determinado tipo de registro de tal forma que, no primeiro caso, um determinado campo é inicializado por uma constante (por exemplo, um nome de variável ou um nome de função com aridade zero) e, no segundo caso, um determinado campo é inicializado por um nome de função com aridade um, ou seja, a função correspondente ao ativo vazio que recebe o tipo de dados do ativo como parâmetro de entrada. Outras formas de adicionar campos são concebíveis, mas essas formas apresentadas revelaram-se suficientes para nosso contexto. Existe também uma operação `addUpdateCase` que estende definições de função para um caso em resposta a um construtor adicionado anteriormente. Diferentes formas de adicionar casos são concebíveis e foi necessária uma forma não trivial no nosso experimento.

A adição de um construtor é necessária para o tipo de dados `TransformationModel`, que é usado em uma função `transform` que, essencialmente, interpreta os modelos de transformação ('termos'). O tipo da função é o seguinte:

```
1 transform :: TransformationModel →SPLModel →InstanceModel →InstanceModel
```

A idéia aqui é que a função tenha os casos discriminados no primeiro argumento (`TransformationModel`) e, essencialmente, a função `transform` seja delegada para uma função de transformação mais específica que prontamente tratará com a transformação dada nos argumentos relacionados ao ativo. Por exemplo, considere uma instância de Hephaestus-PL que suporte ativos casos de uso e processos de negócio. Então, a função `transform` teria a seguinte forma:

```
1 transform :: TransformationModel →SPLModel →InstanceModel →InstanceModel
2 transform (UseCaseTransformation x0) x1 x2 =transformUcm x0 x1 x2
3 transform (BusinessProcessTransformation x0) x1 x2 =transformBpm x0 x1 x2
```

Assim, a operação `addUpdateCase` deve adicionar tais casos que discriminam os casos a serem executados a partir do primeiro argumento e os dois últimos argumentos são definidos pelas variáveis `x1` and `x2` quando passadas para a função no RHS. Uma operação semelhante à `addUpdateCase` foi definida para estender a definição da função `export` em resposta a um construtor adicionado anteriormente no tipo de dados `ExportModel`. A definição da função `export` é a seguinte:

```
1 export :: ExportModel →FilePath → InstanceModel → IO ()
```

Por exemplo, considere uma instância de Hephaestus-PL que deve suportar os ativos casos de uso e processos de negócios em formato de saída xml. Então, a função `export` será sintetizada da seguinte forma:

```
1 export :: ExportModel →FilePath → InstanceModel → IO ()
2 export (ExportUcmXML) x1 x2 =exportUcmToXML (x1 ∪".xml") (ucm x2)
3 export (ExportBpmXML) x1 x2 =exportBpmToXML (x1 ∪".xml") (bpm x2)
```

A extensão da função `xml2Transformation` é também realizada com a operação `addUpdateCase` aplicada em uma lista de casos de transformações suportados pelo CK para o produto emergente de Hephaestus-PL.

Além disso, definimos duas novas operações `addLetInstruction` e `addGeneratorInstruction` para estender a função `main` no módulo Haskell que representa o produto emergente. A primeira é usada para adicionar sentenças `let` na função `main` para recuperar o ativo e para executar a função *parser* do ativo movendo o ativo para a instância Hephaestus-PL. A segunda operação `addGeneratorInstruction` é usada para adicionar instruções sobre os *parsers* do ativo e do CK no módulo do produto emergente. A operação `addLetInstruction` acrescenta a instrução `let product = build fm fc cm spl` responsável pela geração de novas instâncias Hephaestus-PL. Para garantir a compilação dos módulos `HephaestusBase.hs` e `Hephaestus.hs` adicionamos as instruções relacionadas ao CK (declarações `import` e função *parser*) e a chamada da função `build` somente no produto emergente.

Existe também a operação `addUpdateCaseList` baseada na operação `addUpdateCase` que usa uma lista de casos para estender a função `xml2Transformation` que compreende o processo de *parser* do CK realizando o reconhecimento da sintaxe concreta das transformações do ativo na instância Hephaestus-PL.

Finalmente, a operação `removeImportDecl` remove uma declaração `import` do módulo principal do produto emergente de Hephaestus-PL. Esta operação é chamada na transformação `SelectHephaestusBase` para remover a declaração de importação do módulo base de Hephaestus que contém os tipos de dados algébricos. Esta será substituída por uma declaração `import` do módulo com os tipos de dados algébricos do produto emergente Hephaestus-PL.

A operação `removeFunction` remove a função `main` do produto emergente. Isso é necessário na auto geração do *Produto Hephaestus* porque sua função `main` é a função `builHpl` localizada no módulo `IO.hs`. Esta operação é chamada pela transformação `RemoveProductMainFunction` associada à expressão de feature `Hephaestus`.

Tabela 4.3 sumariza o mapeamento das transformações Hephaestus-PL com as operações de metaprogramação.

4.5.4 Módulo Haskell de uma Instância Hephaestus-PL contendo os ativos UCM e BPM

A partir do *Produto Hephaestus* apresentado na Seção 4.4.3, o processo de derivação de produtos gera um novo módulo Haskell que refina o módulo `HephaestusBase` com as features selecionadas da configuração de produto desejada. Por exemplo, a Figura 4.18 mostra o código-fonte da instância Hephaestus-PL gerada pela seleção das features `Use Case`, `Business Process`, `UcmToXML` e `BpmToXML`. Novos casos relacionados à esses ativos

e formatos de saída serão introduzidos na definição das funções `transform` (linhas 17-18) e `export` (linhas 30-31), assim como novos campos de dados serão introduzidos nas definições dos tipos de dados `SPLModel` (linhas 3-4), `InstanceModel` (linhas 9-10), `TransformationModel` (linhas 13-14) and `ExportModel` (linha 27), a função `mkEmptyInstance` (linhas 23-24) que retorna uma instância com novos campos de dados representando os ativos vazios e, finalmente, novos elementos correspondentes ao tipo de dados `ExportModel` são introduzidos na lista `lstExport` (linha 34).

Além disso, o processo de derivação de produtos Hephaestus-PL acrescenta novas sentenças na função `main` para o parser dos ativos `Use Case` and `Business Process` (linhas 39-42) e atualiza os campos da instância `SPLModel` na declaração `spl` (linha 43) e atualiza a declaração `product` (linha 44) inserindo a chamada da função `build`. Além das transformações mencionadas acima que são aplicadas ao módulo `HephaestusBase` durante a derivação do produto, o módulo resultante também precisa receber declarações de *import* para os módulos específicos das features selecionadas no FC.

4.6 Processo Reativo

Como discutido anteriormente, o projeto de Hephaestus-PL deve melhorar a flexibilidade para introduzir suporte ao gerenciamento de variabilidades de novos ativos LPS e assim evoluir o espaço de configurabilidade de Hephaestus-PL. Nesta seção, vamos descrever um processo reativo representado na Figura 4.19 usando a notação BPMN e que poderia orientar os engenheiros de domínio a prosseguir nesta tarefa. É importante notar que nós projetamos este processo com base na nossa experiência em evoluir as versões de Hephaestus para suportar diversos ativos como casos de uso, processos de negócio, requisitos e código. Por isso, focamos no processo reativo para introduzir novos ativos em Hephaestus-PL.

Três papéis contribuem neste processo e estão representados pelas raias na Figura 4.19. Clientes começam o processo reativo em Hephaestus-PL demandando uma ferramenta Hephaestus com novos requisitos. Engenheiros de Aplicação recebem os novos requisitos do cliente, especificam e mapeiam os requisitos para as features de Hephaestus-PL e avaliam se os requisitos são já suportados por Hephaestus-PL. Se sim, eles geram o produto Hephaestus-PL com a configuração desejada, testam-o e entregam-o ao cliente. Se Hephaestus-PL não suporta ainda todos os requisitos da ferramenta solicitada pelo cliente, os engenheiros de aplicação submetem a demanda à análise dos engenheiros de domínio para evoluir Hephaestus-PL para suportar os novos requisitos (ativos LPS).

Os engenheiros de domínio aqui compreendem dois papéis distintos: *especialista de domínio do asset* que define e implementa os artefatos do novo ativo; e *engenheiro de domínio de Hephaestus-PL*, que avalia o impacto e integra os artefatos do novo ativo em Hephaestus-PL. O *especialista de domínio do asset* executa atividades relacionadas ao novo ativo, fornecendo a sua implementação em termos de tipos de dados, transformações, *parsers* e funções de formato de saída. Isso corresponde às atividades: *Define asset type structures*, *Define asset transformations*, *Implement asset parser* e *Implement asset output format* na Figura 4.19. Nenhuma ordem é especificada entre essas atividades porque elas são normalmente realizadas de forma iterativa. O *engenheiro de domínio de Hephaestus-PL* avalia se a introdução de um novo ativo precisa atualizar as APIs do Kernel de Hephaestus-PL (transformações de alto e baixo nível); integra os artefatos do novo

```
data SPLModel =SPLModel {
  featureModel :: FeatureModel,
  splUcm :: UseCaseModel,
  splBpm :: BusinessProcessModel
}

data InstanceModel =InstanceModel {
   featureConfiguration :: FeatureConfiguration ,
  ucm :: UseCaseModel,
  bpm :: BusinessProcessModel
} deriving (Data, Typeable)

data TransformationModel =UseCaseTransformation UseCaseTransformation
                         | BusinessProcessTransformation BusinessProcessTransformation

transform :: TransformationModel →SPLModel →InstanceModel →InstanceModel
transform (UseCaseTransformation x0) x1 x2 =transformUcm x0 x1 x2
transform (BusinessProcessTransformation x0) x1 x2 =transformBpm x0 x1 x2

mkEmptyInstance :: FeatureConfiguration → SPLModel →InstanceModel
mkEmptyInstance fc spl =InstanceModel {
   featureConfiguration =fc ,
  ucm =emptyUcm (splUcm spl),
  bpm =emptyBpm (splBpm spl)
}

data ExportModel =ExportUcmXML |ExportBpmXML

export :: ExportModel →FilePath → InstanceModel → IO ()
export (ExportUcmXML) x1 x2 =exportUcmToXML (x1 ∪".xml") (ucm x2)
export (ExportBpmXML) x1 x2 =exportBpmToXML (x1 ∪".xml") (bpm x2)

lstExport :: [ExportModel]
lstExport =[ExportUcmXML, ExportBpmXML]

main :: IO ()
main =do
 ...
  let bModel   =fromJust (findPropertyValue "businessprocess−model" ps)
  let uModel   =fromJust (findPropertyValue "usecase−model" ps)
  (Core.Success bppl) ∈ parseBusinessProcess (ns bpSchema) (snd bModel)
  (Core.Success ucpl) ∈ parseUseCaseFile (ns ucSchema) (snd uModel)
  let spl =SPLModel{featureModel =fm, splUcm =ucpl, splBpm =bppl}
  let product = build fm fc cm spl
  let out =(outputFile (snd targetDir) (snd name))
  sequence_ [export x out product | x ∈ lstExport]
```

Figura 4.18: Trecho de Código da instância Hephaestus-PL contendo os ativos UCM e BPM com seus formatos de saída XML.

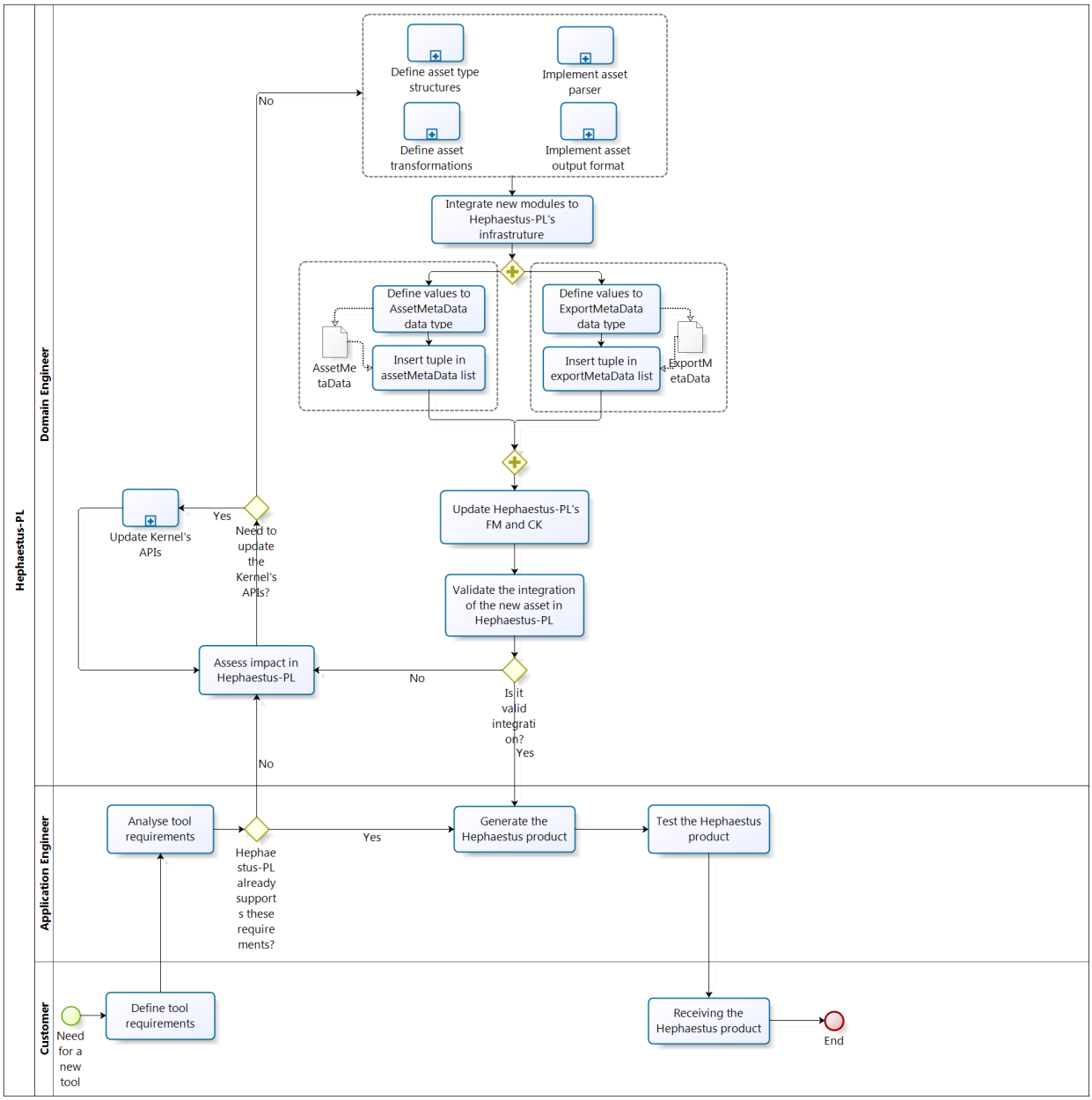

Figura 4.19: Processo Reativo para introduzir novos ativos em Hephaestus-PL.

ativo na infraestrutura de Hephaestus-PL; atualiza as estruturas de metadados inserindo referências para o novo ativo; atualiza o FM e CK de Hephaestus-PL usando o guia dos templates de evolução segura de linhas de produtos de software definidos em (Neves et al., 2011); e valida o novo ativo gerando um novo produto Hephaestus-PL com o novo ativo selecionado e verificando a corretude do código-fonte gerado. Se o produto gerado é uma instância válida de Hephaestus-PL, então as atividades do engenheiro de domínio estão finalizadas.

4.6.1 Evolução de Hephaestus-PL para Suportar o Ativo Requisitos

Para ilustrar a execução do processo reativo, nós apresentamos a evolução de Hephaestus-PL para suportar o gerenciamento de variabilidades no ativo *Requirement Model*. A versão atual de Hephaestus-PL suporta os ativos *Use Case Model* e *Business Process Model*. Figura 4.20 resume este cenário de evolução de Hephaestus-PL que descrevemos abaixo.

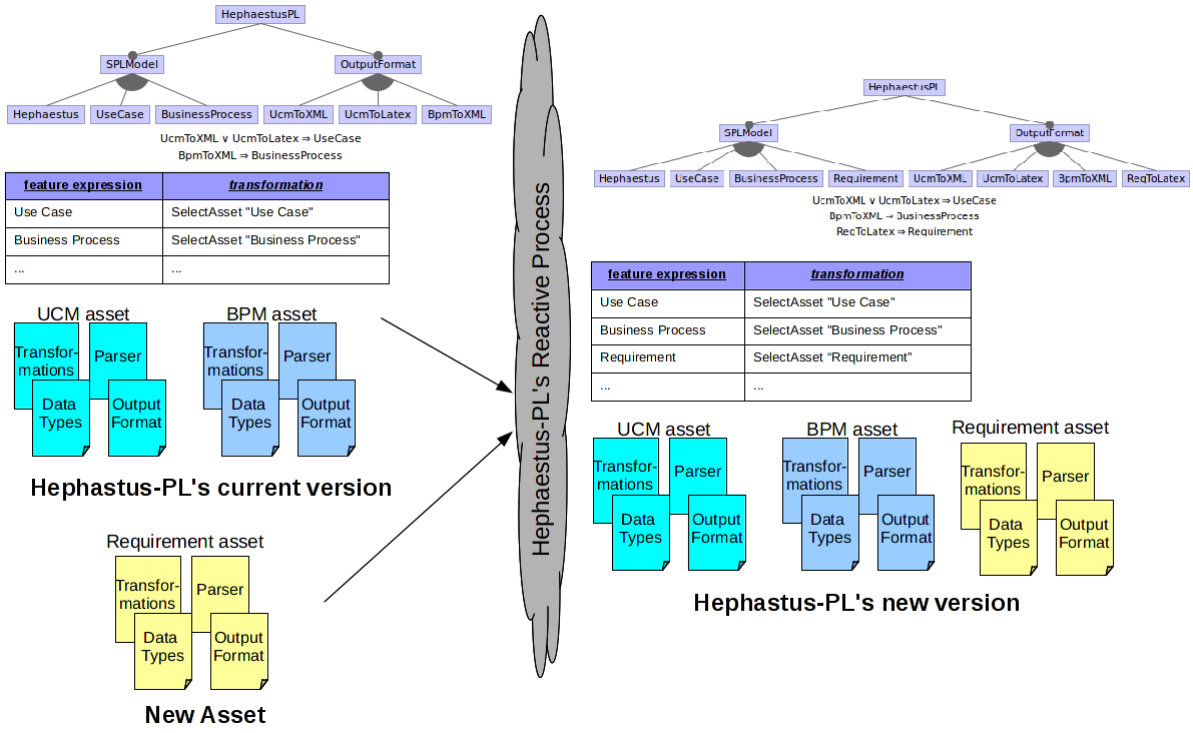

Figura 4.20: Evolução de Hephaestus-PL para introduzir o ativo *Requirement*.

O processo reativo para introduzir novos ativos em Hephaestus-PL começa com o *engenheiro de domínio* avaliando os impactos nas APIs do kernel de Hephaestus-PL para suportar o ativo *Requirement*. É necessário analisar se as APIs de alto nível – correspondentes às transformações Hephaestus-PL (*SelectAsset* e *SelectExport*, principalmente) – e as APIs de baixo nível – correspondentes às operações de metaprogramação – suportam a especificação do novo ativo *Requirement*. Este processo reativo considera que as transformações Hephaestus-PL (*SelectAsset* e *SelectExport*) suportam novos ativos, ou

```
1 data Requirement =Requirement {
2         reqId  ::  Id ,
3         reqName :: String ,
4         reqDescription  ::  String
5 } deriving (Show, Data, Typeable)
6
7 data RequirementModel =RM {
8         reqs  ::  [Requirement]
9 } deriving (Show, Eq, Data, Typeable)
```

Figura 4.21: Definição do tipo de dados `RequirementModel`

seja, com base em nossa experiência, introduzir suporte à variação em um novo ativo em Hephaestus-PL não tem impacto sobre as APIs de alto nível.

Em seguida, o *especialista de domínio do asset* deve implementar os quatro artefatos do ativo *Requirement* para evoluir Hephaestus-PL dando suporte ao novo ativo. Os itens abaixo descrevem o que é necessário definir para o novo ativo *Requirement*:

(i) definir o tipo de dados `RequirementModel` e outros tipos auxiliares que representam um conjunto de requisitos com suas variabilidades. O tipo de dados `RequirementModel` é composto pelos campos id, nome e descrição (ver trecho de código na Figura 4.21) e está localizado em um módulo específico, ou seja, o módulo `HplAssets.ReqModel.Types.hs`;

(ii) definir as transformações e a função instância vazia do ativo *Requirement*. As transformações especificadas que gerenciam as variabilidades em *Requirement* são `SelectAllRequirements`, `SelectRequirements` e `RemoveRequirements`. `emptyReq` é a função que define uma instância vazia do ativo *Requirement*. Também é necessário definir um tipo de dados e uma função que compreendam todas as transformações do ativo *Requirement* para introduzí-los em uma instância de produto Hephaestus-PL. Por exemplo, nós definimos o tipo de dados `RequirementTransformation` e a função `transformReq`, respectivamente, e apresentados na Figura 4.22. A definição `RequirementTransformation` deve ter a cláusula `deriving (Show, Eq, Ord)` para permitir a ordenação e a visualização das transformações do ativo *Requirement*. Todas as definições de tipos de dados para o ativo *Requirement* devem estar no módulo `HplAssets.ReqModel.Types.hs`. A função `transformReq` deve ter uma assinatura similar à função `transform` do módulo `HephaestusBase.hs` que é uma instância base de Hephaestus. Além disso, a função `transformReq` deve ser construída por *pattern matching* em Haskell tendo um novo caso para cada construtor declarado no tipo de dados `RequirementTransformation` (ver trecho de código na Figura 4.22). Ambas as funções `emptyReq` and `transformReq` são implementadas em um módulo específico, ou seja, o módulo `HplAssets.Requirements.hs`. Essas funções devem ser visíveis pelo kernel de Hephaestus-PL para que ele seja capaz de gerar uma nova instância de Hephaestus-PLcom o ativo *Requirement* selecionado;

(iii) implementar uma nova função parser do ativo *Requirement* para a leitura dos artefatos da linha de produtos *Requirement* de um formato de representação pública, como XML, para o formato interno de Hephaestus-PL (tipo de dado `RequirementModel`).

(iv) implementar uma nova função de formato de saída do ativo *Requirement*, ou seja, a função `exportReqToLatex` que permite a exportação de instâncias do produto *Requirement* no formato de saída LaTeX para fora da ferramenta Hephaestus-PL.

```
1  data RequirementTransformation =SelectAllRequirements
2                                  | SelectRequirements [Id]
3                                  | RemoveRequirements [Id]
4                                  deriving (Show, Eq, Ord)
5
6  emptyReq :: RequirementModel →RequirementModel
7  emptyReq reqmodel =reqmodel { reqs =[] }
8
9  transformReq :: RequirementTransformation
10              → SPLModel
11              → InstanceModel
12              → InstanceModel
13
14 transformReq (SelectAllRequirements) spl product =...
15 transformReq (SelectRequirements ids) spl product =...
16 transformReq (RemoveRequirements ids) spl product =...
```

Figura 4.22: Definição das transformações do `RequirementModel` e função `emptyReq`

Depois de definir os quatro artefatos do ativo *Requirement*, o engenheiro de domínio realiza a integração desses módulos do novo ativo *Requirement* na estrutura de diretórios de Hephaestus-PL. É criado um novo diretório para o ativo *Requirement* abaixo do diretório `HplAssets`, ou seja, o diretório `ReqModel` onde são colocados os módulos do novo ativo, exceto o módulo `HplAssets.Requirements.hs` com as definições das funções `emptyReq` e `transformReq` que é colocado no mesmo nível do diretório `ReqModel` (abaixo do diretório `HplAssets`).

Em seguida, o engenheiro de domínio utiliza algumas informações contidas nos módulos do ativo *Requirement* e define os conjuntos de dados que suportam a execução das APIs de baixo nível do kernel de Hephaestus-PL. Isso representa estender as estruturas de metadados `AssetMetaData` e `ExportMetaData` definidas no módulo `HplAssets.Hephaestus.MetaData.hs` para suportar o novo ativo *Requirement*. Algumas informações definidas nos módulos do ativo *Requirement*, tais como identificadores de tipos de dados, campos de dados, funções e módulos são inseridas nas estruturas de metadados. Essas informações serão utilizadas pelas operações de metaprogramação para estender os tipos de dados e funções *abertos* no produto base de Hephaestus e gerar uma instância de Hephaestus-PL com o ativo *Requirement* selecionado. Por exemplo, informações sobre o ativo *Requirement* tais como o nome do campo de dados e o tipo de dados para estender os tipos de dados *SPLModel* e *InstanceModel*, ou seja, `("splReq","RequirementModel")` e `("req","RequirementModel")`, respectivamente; o nome da função que define uma instância vazia do ativo *Requirement*, ou seja, `emptyReq`; o nome da função que implementa o *parser* do ativo *Requirement*, ou seja, `parseRequirementModel`; o nome do tipo de dados que define todas as transformações do ativo *Requirement*, ou seja, `RequirementTransformation`.

Depois disso, o engenheiro de domínio atualiza o FM e CK de Hephaestus-PL definidos no módulo `HplAssets.Hephaestus.MetaData.hs` usando os templates de evolução segura de linhas de produtos de software definidos em (Neves et al., 2011). As novas

or-features correspondentes ao ativo *Requirement* e seu formato de saída LaTeX (feature *ReqToLatex*) e a nova restrição global do FM $ReqToLatex \Rightarrow Requirement$ são inseridas no FM de Hephaestus-PL. Introduzimos a nova *or-feature Requirement* abaixo da feature mandatória *SPLModel* e introduzimos a nova *or-feature ReqToLatex* abaixo da feature opcional *OutputFormat* (ver Figura 4.23). O CK de Hephaestus-PL é atualizado com o novo mapeamento de expressões de features para transformações sobre o ativo *Requirement* conforme apresentado na Tabela 4.4.

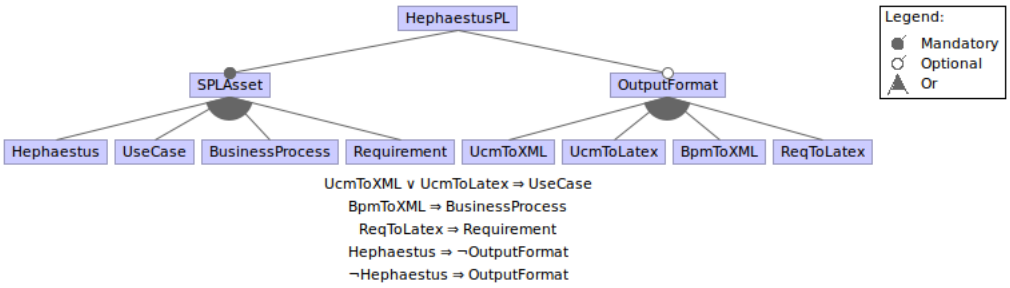

Figura 4.23: FM de Hephaestus-PL depois de introduzir o ativo *Requirement*.

Expressões de Features	**Transformações**
True	SelectHephaestusBase
Hephaestus	BindProductName ”Hephaestus” SelectAsset ”Hephaestus” RemoveProductMainFunction
NOT Hephaestus	SelectCKParser
UseCase	SelectAsset ”Use Case”
UseCase AND UcmToXML	SelectExport ”UcmToXML”
UseCase AND UcmToLatex	SelectExport ”UcmToLatex”
BusinessProcess	SelectAsset ”Business Process”
BusinessProcess AND BpmToXML	SelectExport ”BpmToXML”
Requirement	SelectAsset ”Requirement”
Requirement AND ReqToLatex	SelectExport ”ReqToLatex”

Tabela 4.4: CK de Hephaestus-PL depois de introduzir o ativo *Requirement*.

Finalmente, o engenheiro de domínio gera uma instância de produto Hephaestus-PL selecionando uma configuração de produto somente com o novo ativo *Requirement* integrado em Hephaestus-PL. Ele valida se a instância gerada contém as definições corretas do ativo *Requirement* nos pontos de variabilidade do produto base de Hephaestus e reporta a nova versão de Hephaestus-PL para o engenheiro de aplicação. Suponha que a validação da integração do ativo *Requirement* em Hephaestus-PL não foi bem sucedida, então é necessário retornar para a atividade de avaliação de impacto nas APIs do kernel de Hephaestus-PL até obter a validação correta da instância de Hephaestus-PL gerada.

4.7 Resultados e Discussões

A fim de orientar a discussão dos resultados, aplicamos o método *Goal Question Metric* (GQM) (Basili et al., 1994) para estruturar o contexto, o objeto de estudo, suas propriedades, o objetivo e como este último pode ser operacionalizado e respondido. Nesta seção vamos primeiro discutir os objetivos, questões e métricas de nossa investigação (Subseção 4.7.1) e, em seguida, vamos avaliar nosso estudo empírico (Subseção 4.7.2) e, finalmente, discutiremos suas limitações (Seção 4.7.3).

4.7.1 Objetivos, Questões e Métricas

Nosso objetivo é avaliar a configurabilidade e flexibilidade de Hephaestus-PL (Seções 4.4 and 4.5) relativo à fase de engenharia de aplicação para o desenvolvimento de produtos de software e dentro do contexto de diferentes artefatos. Desde que Hephaestus-PL surgiu de uma evolução sistemática de Hephaestus, este também é avaliado como uma linha de base (*baseline*). A Tabela 4.5 resume o objetivo geral de avaliação do nosso trabalho.

Propósito	avaliar
Questão	a configurabilidade e flexibilidade de
Objeto	Hephaestus-PL e Hephaestus
Ponto de Vista	perspectiva de engenharia de aplicação
Contexto	diferentes artefatos

Tabela 4.5: Objetivo GQM do artigo II.

Na Seção 4.2 nós caracterizamos o *contexto*, o *ponto de vista* e parte do *objeto* do projeto inicial de Hephaestus, que foi projetado para gerenciar a variabilidade em cenários de casos de uso, e sua evolução para suportar variabilidade em outros ativos. Nas Seções 4.4 e 4.5 detalhamos a outra parte do *objeto*, ou seja, Hephaestus-PL. Esta seção prossegue com a análise de Hephaestus-PL e Hephaestus, utilizando uma avaliação qualitativa e quantitativa que responde às nossas métricas de GQM satisfazendo assim o *propósito* e a *questão* do objetivo do estudo.

A partir do objetivo do estudo apresentado na Tabela 4.5, derivamos perguntas que melhor caracterizam o nosso estudo. Além disso, relacionada a cada pergunta, usamos uma ou mais métricas quantitativas ou qualitativas para indicar o nível de conformidade das ferramentas Hephaestus e Hephaestus-PL em relação ao objetivo do estudo. "Métricas" de avaliação qualitativa são também concebidas no método GQM (Basili et al., 1994). A seguir, apresentamos as questões (**Q**) e as métricas (**M**) do nosso modelo GQM e explicamos como elas representam e guiam para o objetivo do estudo.

Q1 *É o mecanismo de variabilidade utilizado suficientemente expressivo?*

- Métrica **M1.1**: Existe suporte para tipos de dados abertos?
- Métrica **M1.2**: Existe suporte para funções abertas?
- Métrica **M1.3**: Existe suporte para a instanciação de um ativo LPS?

- Métrica **M1.4**: Existe suporte para a composição de ativos sem ter que instanciar todos os ativos (sem usar *over features*)?

Q2 *Dado um número fixo de ativos, qual é o esforço para endereçar uma nova configuração?*

- Métrica **M2.1**: Número de módulos alterados.
- Métrica **M2.2**: Número de linhas de código alteradas.
- Métrica **M2.3**: Número de módulos criados.
- Métrica **M2.4**: Número de linhas de código criadas.
- Métrica **M2.5**: Existe suporte à automação?
- Métrica **M2.6**: Qual é a complexidade analítica do esforço?

Q3 *Qual é o esforço para endereçar um novo ativo?*

- Métrica **M3.1**: Número de módulos alterados.
- Métrica **M3.2**: Número de artefatos de código alterados.
- Métrica **M3.3**: Número de módulos criados.
- Métrica **M3.4**: Número de artefatos de código criados.
- Métrica **M3.5**: Existe suporte à automação?

Q4 *É suportado o gerenciamento modular do ativo relacionado à sua variabilidade?*

- Métrica **M4.1**: As alterações relacionadas à variabilidade do ativo LPS são homogêneas ou heterogêneas?
- Métrica **M4.2**: As alterações relacionadas à variabilidade do ativo LPS são localizadas ou espalhadas?
- Métrica **M4.3**: Qual é a possibilidade de automação das alterações relacionadas com a variabilidade dos ativos LPS?

Questões Q1-Q4 representam características relevantes do desenvolvimento de Hephaestus-PL quando comparado com Hephaestus. Q1 remete para a questão da configurabilidade endereçando a expressividade necessária do mecanismo de gerenciamento de variabilidade requerido. Correspondentemente, as métricas M1.1, M1.2, M1.3 e M1.4 investigam se existe suporte para os tipos de variabilidades em Hephaestus-PL classificados como tipos de dados abertos, funções abertas, instanciação de um único ativo e composição de ativos, como explicado na Seção 4.3. Em particular, a métrica M1.4 foca na questão atual de Hephaestus: a necessidade de instanciar um produto com mais de um ativo LPS definido nos tipos de dados algébricos `SPLModel` e `InstanceModel`. Da mesma forma, Q2 também trata a questão da configurabilidade, mas endereça-a dentro de um escopo pré-definido de ativos. As métricas de Q2 abordam o esforço para adicionar uma nova configuração a partir de diferentes perspectivas de granularidade (módulos e linhas de código), suporte à automação e complexidade analítica, sendo esta última relevante para a escalabilidade. Diferentemente, Q3 rastreia a questão da flexibilidade por investigar o esforço de evolução necessário para endereçar a variabilidade em um novo ativo. Esta questão é refinada por métricas a partir de diferentes perspectivas de granularidade onde artefatos de código

representam tipos de dados e funções, exceto módulos, e considerando também o suporte à automação. Finalmente, Q4 também se refere à questão de flexibilidade e avalia se a modularidade no tratamento da variabilidade do ativo é suportada ou não, uma propriedade importante para a abordagem reativa no desenvolvimento de LPS uma vez que ela suporta potencialmente a introdução de novas funcionalidades. A homogeneidade (métrica M4.1) refere-se aos tipos de estruturas sintáticas Haskell (tipos de dados, funções, classes) associados com as alterações advindas da resolução das variabilidades dos ativos. Consideramos *homogênea* quando existe apenas um tipo de estrutura sintática para alterar e classificamos de *heterogênea*, em caso contrário. A localidade das alterações (métrica M4.2) é avaliada nos módulos com código entrelaçado de features e pode ser definida como *localizada* quando todo o código da feature está em apenas um módulo ou *espalhada*, em caso contrário. Por fim, a métrica M4.3 que representa a viabilidade de automação das alterações de código associadas às features, é definida pela derivação das métricas M4.1 e M4.2. Assim, M4.3 é classificada em *alta* quando observamos os valores *homogênea* para M4.1 e *localizada* para M4.2. Qualquer valor diferente em M4.1 e M4.2, definimos M4.3 como *baixa*.

4.7.2 Avaliação

Dada a configuração GQM da Subseção 4.7.1, vamos proceder agora à avaliação das técnicas de projeto e implementação para o gerenciamento da variabilidade (Seções 4.4 e 4.5). Tabela 4.6 resume a avaliação para Hephaestus e Hephaestus-PL das métricas associadas às questões definidas no nosso modelo GQM.

Métrica	**Hephaestus**	**Hephaestus-PL**
M1.1	Não	Sim
M1.2	Não	Sim
M1.3	Sim	Sim
M1.4	Não	Sim
M2.1	1	0
M2.2	9	0
M2.3	0	0
M2.4	8	0
M2.5	Não	Sim
M2.6	$O(n)$	$O(k)$
M3.1	5	1
M3.2	6	4
M3.3	4	4
M3.4	baixo, depende do novo ativo	baixo, depende do novo ativo
M3.5	Não	Não
M4.1	heterogêneas	homogêneas
M4.2	espalhadas	localizadas
M4.3	baixa	alta

Tabela 4.6: Resumo da avaliação das métricas do modelo GQM

Com relação à Q1, quando avaliamos Hephaestus-PL existe expressividade suficiente para suportar todos os tipos de variabilidade de Hephaestus. O suporte à variabilidade em tipos de dados e funções (métricas M1.1 e M1.2) é garantido pela abordagem transformacional para o gerenciamento da variabilidade utilizada em Hephaestus-PL, ou seja, usando operações de metaprogramação para estender os tipos de dados e funções. O suporte para a composição de ativos, ou seja, a instanciação de um produto com um ou mais ativos (métricas M1.3 e M1.4) é garantida pelo processo `build` do kernel de Hephaestus-PL e operações de metaprogramação gerando uma variante de Hephaestus-PL que representa uma instância de produto que combina quaisquer ativos do modelo de features de Hephaestus-PL. Quando avaliamos Hephaestus como uma variante gerada manualmente que oferece suporte à derivação de produtos utilizando uma abordagem composicional de artefatos e transformações a partir do CK para gerar um novo produto, observamos que não há suporte para o gerenciamento de variabilidades em tipos de dados e funções pela introdução de novos ativos e para a composição de quaisquer ativos. Relacionado à métrica M1.4, a geração de uma variante Hephaestus que suporte uma composição de ativos sem ter que instanciar todos os ativos exige um esforço considerável porque é necessário alterar vários artefatos de código (módulos, tipos de dados e funções). Esta mudança *ad hoc* em diferentes partes do código de Hephaestus é uma atividade propensa à erros e não é geralmente realizada dessa forma. Alternativamente, é utilizado o conceito de *Over Feature* (Batory et al., 2000) para gerar novas variantes Hephaestus porque o impacto é considerado pequeno e limitado na função `main`.

Quanto à Q2, quando avaliamos Hephaestus-PL, o esforço para endereçar uma nova configuração a partir de um número fixo de ativos é constante e pequeno e representa o esforço para especificar a nova configuração de produto para Hephaestus-PL contendo os ativos selecionados, considerando que os ativos da configuração desejada já estão integrados em Hephaestus-PL. Portanto, a medida para a métrica M2.6 é constante e próxima de zero ($\simeq O(k)$). Da mesma forma, a medida para as métricas M2.1, M2.2 e M2.3 e M2.4 é zero, ou seja, não temos módulos criados nem modificados. Em relação à métrica M2.5, há um suporte para a automação em Hephaestus-PL para gerar novas configurações de ativos baseado no processo `build`, nas transformações Hephaestus-PL e nas operações de metaprogramação que tratam a variabilidade em Hephaestus-PL.

Quando avaliamos Hephaestus com relação à Q2, o esforço para endereçar uma nova configuração de ativos pode ser visto de duas maneiras: usando ou não o conceito de *Over Feature*. Primeiro, usando o conceito de *Over Feature* e um produto Hephaestus que suporta um conjunto de ativos onde a configuração desejada é um subconjunto desses ativos, o esforço é considerado pequeno com impacto apenas sobre o módulo `Main.hs` que precisa de algumas mudanças no código. Portanto, a medida da métrica M2.1 é um módulo e a medida da métrica M2.3 é zero módulo (isto é, nenhum módulo é criado). Linhas de código relacionadas aos ativos selecionados são alteradas e criadas no módulo `Main.hs`, ou seja, importar os módulos de tipos de dados algébricos, módulos das funções *parser* e formato de saída, leitura do arquivo do ativo LPS no arquivo de propriedades de entrada de Hephaestus, chamada do *parser* do ativo, atualização da função `createSPL` após executar o *parser* do ativo e executar a chamada da função de exportação do ativo no formato de saída. Isso representa cerca de nove linhas alteradas (métrica M2.2) e oito linhas criadas (métrica M2.4) no módulo `Main.hs`. Com relação à métrica M2.5, não há suporte à automação em Hephaestus e a complexidade analítica do esforço (métrica

M2.6) utilizando o conceito de *Over Feature* em Hephaestus é a soma do número de linhas alteradas e criadas (métricas M2.2 e M2.4) multiplicado pelo número de ativos (n) da configuração desejada, ou seja, $(M2.2 + M2.4) * n \simeq O(n)$. Portanto, o esforço é proporcionalmente linear ao número de ativos da configuração.

Analisando agora o esforço para endereçar uma nova configuração de ativos quando não usamos o conceito de *Over Feature* em Hephaestus, observamos que o impacto é bastante alto porque é necessário introduzir na variante de Hephaestus o código relacionado aos ativos selecionados e isso requer um bom entendimento do código e mudanças em vários módulos, tipos de dados e funções. Além disso, por ser uma atividade manual é mais propensa à erros e para verificar as configurações inválidas não é uma tarefa fácil de ser feita manualmente. Assim, para usarmos apenas o código dos ativos da configuração de produto selecionada em Hephaestus, seria necessário copiar o código fonte da variante Hephaestus para um novo diretório e remover as features não selecionadas. Atualmente, Hephaestus não funciona assim.

Quanto à questão Q3 que refere-se ao processo reativo, em ambas as ferramentas Hephaestus-PL e Hephaestus o esforço para endereçar um novo ativo considera que o esforço inicial e maior é para a geração dos elementos do novo ativo, tais como os tipos de dados abstratos, as transformações, a função *parser* e a função de formato de saída. Este esforço é o mesmo em ambas as ferramentas e sua avaliação está fora do escopo deste artigo. Após a geração dos elementos do ativo, existe o esforço para integrar estes elementos do novo ativo em Hephaestus-PL e Hephaestus e é esse esforço que vai realmente interessar nesta nossa avaliação das métricas de Q3.

Quando avaliamos Hephaestus-PL com relação às métricas M3.1 e M3.2, temos apenas um módulo alterado (`MetaData.hs`) e quatro artefatos de código alterados. No módulo `MetaData.hs` precisamos mudar as funções `featuremodel` and `configurationKnowledge` que definem o FM e CK do Hephaestus-PL, e mudar as listas `assetMetaData` and `exportMetaData` que contêm as estruturas de metadados que suportam as operações de metaprogramação. Com relação às métricas M3.3 e M3.4, temos quatro módulos criados: `Types.hs` contém os tipos de dados algébricos, `<NewAsset>.hs` contém as transformações que gerenciam as variabilidades do novo ativo, `<NewAssetFomartParser>.hs` contém a função de *parser* do ativo e `<NewAssetOutputFormat>.hs` contém as funções de formato de saída do ativo; e a soma de artefatos de código criados (tipos de dados e funções) varia de acordo com o ativo. Atualmente, não há suporte de automação em Hephaestus-PL para endereçar um novo ativo (métrica M3.5), mas é possível implementar futuramente utilizando um conjunto mínimo de variáveis e um processo de inferência automática ou semi-automática das informações de metadados do ativo nos módulos do ativo desde que as estruturas de metadados tenham uma estrutura regular e a implementação do ativo siga determinadas regras de projeto (*design rules*) para permitir essa inferência automática ou semi-automática.

Quando avaliamos Hephaestus com relação à métrica M3.1, contabilizamos cinco módulos alterados que correspondem aos módulos dos tipos de dados do CK, interpretador do CK, *parser* XML do CK, exportação de produtos e módulo principal de Hephaestus. Com relação à métrica M3.2, são seis artefatos de código alterados que correspondem aos tipos de dados `SPLModel` e `InstanceModel` e às funções `build`, `xml2Transformation`, `exportProduct` e `main`. Com relação à métrica M3.3, assim como ocorre em Hephaestus-PL, são também quatro módulos criados que correspondem à definição dos tipos de dados,

transformações, *parser* e formato de saída do novo ativo. A medida da métrica M3.4 em Hephaestus que corresponde ao número de artefatos de código criados para representar os elementos do novo ativo (tipos abstratos de dados, transformações, função *parser* e função de formato de saída) varia de acordo com o ativo. Em ambos Hephaestus-PL e Hephaestus o número de artefatos de código alterados e criados é pequeno porque são implementados em linguagem Haskell que é uma linguagem funcional declarativa. Em Hephaestus não há suporte de automação para endereçar um novo ativo (métrica M3.5) e é mais difícil e complexo tentar implementar alguma automação porque em Hephaestus o código do ativo está espalhado em cinco módulos e seu tipo de variabilidade é heterogêneo.

Em termos de modularização de features (Q4), observamos que ambos Hephaestus-PL e Hephaestus têm suporte à modularidade nos elementos do ativo LPS que são definidos em módulos independentes e têm a sua implementação totalmente confinada nos módulos específicos, como os quatro módulos que definem, respectivamente, os tipos de dados algébricos, transformações, *parser* e formato de saída do ativo. No entanto, existe código associado ao ativo LPS que está espalhado e entrelaçado com o código de outros ativos LPS em alguns módulos. Portanto, para avaliar se o gerenciamento do ativo LPS relacionado à sua variabilidade é modular, definimos três métricas. Nós avaliamos a homogeneidade (métrica M4.1), a localidade (métrica M4.2) e a possibilidade de automação (métrica M4.3) das mudanças relacionadas aos pontos de variação do ativo LPS nos objetos Hephaestus-PL e Hephaestus como critérios qualitativos para a questão Q4.

Em Hephaestus definimos as mudanças para endereçar um ativo LPS como heterogênea, espalhada e, portanto, com baixa possibilidade de automação. Elas são heterogêneas porque se referem à diferentes tipos de alterações, tais como inserção de campos em tipos de dados, definição de novas instâncias de classe, definição de novas funções e introdução de novas sentenças em funções. As mudanças são espalhadas porque ocorrem em cinco diferentes módulos de Hephaestus (conforme apresentado na métrica M3.1 de Hephaestus) e os módulos de Hephaestus que endereçam a variabilidade do ativo LPS precisam ser atualizados manualmente e isto representa uma atividade propensa à erros. Ou seja, as mudanças são heterogêneas, espalhadas em vários módulos e entrelaçadas com outros ativos LPS.

Por outro lado, em Hephaestus-PL as alterações para endereçar novos ativos são classificadas como homogêneas e localizadas, uma vez que elas apenas se referem à definição das informações de metadados do ativo LPS em um único módulo `MetaData.hs`. Assim, em Hephaestus-PL há mais possibilidade de automação usando um processo de inferência nos módulos do ativo LPS implementados de acordo com *design rules* previamente definidas. Tivemos um esforço inicial maior no desenvolvimento de Hephaestus-PL com a participação de um especialista em linguagem Haskell para a criação da infraestrutura inicial de Hephaestus-PL que contribuiu para um esforço menor para endereçar as mudanças relacionadas aos novos ativos quando comparado ao esforço para o mesmo propósito observado em Hephaestus.

Além disso, em Hephaestus-PL algum grau de modularidade foi obtido pelo mapeamento das configurações de produtos de Hephaestus-PL em transformações usando metaprogramação, onde a variabilidade de Hephaestus-PL relacionada com a configuração dos ativos é tratada dinamicamente e automaticamente na geração de uma variante Hephaestus-PL. Isso é permitido pelo processo transformacional de geração de produtos em Hephaestus-PL com o suporte das operações de metaprogramação que atendem às

necessidades de gerenciamento da variabilidade dos ativos nos artefatos tipos de dados abertos e funções abertas do módulo base de Hephaestus.

4.7.3 Ameaças à Validade

As ameaças à validade do nosso estudo podem ser apresentadas da seguinte forma:

- *Validade de Construção* refere-se ao estabelecimento de medidas operacionais corretas para os conceitos que estão sendo estudados. As principais construções em nosso estudo são os conceitos de "configurabilidade" e "flexibilidade". Em relação ao primeiro, usamos métricas indiretas considerando a variabilidade e o esforço de adicionar uma configuração. Quanto à segunda construção, usamos métricas indiretas considerando o esforço de evolução para endereçar um novo ativo, bem como uma característica chave interna, a modularidade. Apesar do esforço não ter sido computado em tempo, o esforço foi baseado em métricas indiretas como as apresentadas para a questão Q2. O uso de experimentos como trabalhos futuros pode melhorar essa avaliação.
- *Validade Interna* trata em estabelecer uma relação de causalidade em que certas condições são mostrados para levar a outras condições. No caso, as decisões de projeto por trás da definição de arquitetura rastreiam para resolver questões de variabilidade demandadas pela melhoria do suporte à configurabilidade. O processo reativo proposto foi concebido para abordar a questão da flexibilidade. Entretanto, as métricas foram coletadas de forma manual, o que é suscetível a erros.
- *Validade Externa* diz respeito em estabelecer o domínio para que os resultados de um estudo possam ser generalizados. Embora, o nosso estudo concentra-se em uma única ferramenta Hephaestus acreditamos que a abordagem e conceitos do projeto de Hephaestus-PL e o processo reativo suportado podem ser usados para melhorar a configurabilidade e flexibilidade em outras ferramentas de derivação de produtos de LPS desenvolvidas usando o mesmo paradigma de linguagens funcionais. Para ferramentas desenvolvidas em linguagens procedurais pode ser mais complicado estender a funcionalidade de uma função. Então, para esses casos é necessário realizar uma análise mais detalhada da aplicabilidade e validade da nossa solução.
- *Confiabilidade* se preocupa em demonstrar que as operações de um estudo podem ser repetidas com os mesmos resultados. Dados o projeto, a implementação e descrições do processo reativo esperamos que repetições do nosso estudo ofereçam resultados similares aos nossos. Para isso, vamos disponibilizar os artefatos do nosso projeto e implementação de Hephaestus-PL para que os experimentos sejam replicados.

4.8 Trabalhos Relacionados

Trabalho anterior discute a necessidade de endereçar o desenvolvimento de ferramentas de LPS como próprio desenvolvimento de LPS (Grünbacher et al., 2008). No entanto, as orientações detalhadas sobre a engenharia de domínio e engenharia de aplicação não são exploradas.

Conforme discutido na Seção 4.3, um requisito fundamental de Hephaestus-PL é suportar qualquer configurabilidade de artefatos de produtos, mas isso não tem sido explicitamente abordado em nenhum trabalho ainda.

Em (Batory et al., 2003) foi realizado o *bootstrapping* de AHEAD a partir de AHEAD. Isso é semelhante ao nosso trabalho, onde realizamos o *bootstrapping* de Hephaestus a partir do próprio Hephaestus. Entretanto, (Batory et al., 2003) não foca em atender diferentes artefatos e não oferece suporte explícito para *or-features*. Além disso, a linguagem de programação e paradigma utilizados são diferentes do nosso.

Transkell (Xavier e Borba, 2010) é uma linguagem de domínio específico (DSL) desenvolvida para extrair e modularizar as variabilidades da ferramenta Hephaestus e torná-la uma linha de produtos. É uma técnica de abordagem transformacional similar à técnica usada no nosso trabalho. Em Transkell, o modelo de features de Hephaestus representa as transformações de Hephaestus. A sintaxe e semântica da linguagem Transkell foram implementadas no ambiente Stratego/XT (Visser, 2003). Em nosso trabalho nos concentramos nas variabilidades de ativos que representam os diferentes artefatos de domínio de Hephaestus e apresentamos os detalhes de Hephaestus-PL desenvolvido como uma linha de produtos.

A abordagem *DOPLER (Decision-Oriented Product Line Engineering for effective Reuse)* (Dhungana et al., 2011) representa um conjunto de ferramentas para desenvolvimento de ferramentas de LPS como próprio desenvolvimento LPS (Grünbacher et al., 2008). Essa solução é composta por *DoplerVML*, uma linguagem de modelagem de variabilidades para definir linhas de produtos baseada em modelos de decisão com ênfase na derivação de produtos. *DOPLER* foi inicialmente desenvolvido para suportar o domínio de automação industrial (*Siemens VAI*), mas a proposta é ser extensível e personalizável à diferentes contextos para atender as necessidades de diferentes usuários e organizações. No entanto, a abordagem *DOPLER* não suporta configurabilidade de qualquer combinação de ativos, como nós propomos na solução do nosso trabalho.

Alguns outros estudos comparativos com ferramentas de derivação de produtos foram conduzidos (Torres et al., 2010b,a). (Torres et al., 2010b) analisou seis modernas ferramentas de derivação de produtos (*Captor*, *CIDE*, *GenArch*, *Hephaestus*, *pure::variants* e *XVCL*) no contexto de cenários de evolução de uma linha de produtos de software. O estudo analisou a modularidade, complexidade e estabilidade de artefatos de derivação de produtos ao longo da evolução de uma linha de produtos móvel. A avaliação mostrou que a modularidade e estabilidade em linhas de produtos de software são favoráveis à flexibilidade da ferramenta. Em (Torres et al., 2010a) a flexibilidade e extensibilidade de ferramentas de desenvolvimento de linhas de produtos que suportam a derivação de produtos baseadas em abordagens DDM e DSOA, foram avaliadas quanto ao suporte à adição de novas funcionalidades.

4.9 Conclusões

Hephaestus-PL é uma linha de produtos resultante da evolução da ferramenta Hephaestus. Hephaestus (Bonifácio et al., 2009) foi inicialmente projetada para suportar a engenharia de aplicação e o gerenciamento de variabilidade em uma linha de produtos de cenários de casos de uso. Com o tempo, Hephaestus evoluiu para gerenciar a variabilidade em outros ativos, tais como requisitos, código fonte e processos de negócio. No entanto,

essa ferramenta não foi projetada com flexibilidade e configurabilidade adequadas para permitir a sua customização para endereçar a variabilidade em um novo ativo específico nem em qualquer combinação desejada de ativos.

Para suprir esta deficiência em Hephaestus, Hephaestus-PL foi desenvolvido. Hephaestus-PL aumenta a configurabilidade de Hephaestus permitindo a derivação de instâncias da ferramenta que gerenciam variabilidade em qualquer combinação de ativos. Além disso, sua flexibilidade permite a extensão sistemática de Hephaestus-PL para acrescentar novos ativos e suas combinações. Uma vez que Hephaestus-PL foi gerado a partir do *bootstrapping* de Hephaestus, definimos uma abordagem reativa para aumentar a sua configurabilidade e alcançar o objetivo de permitir a geração de diferentes instâncias de Hephaestus.

Uma avaliação revela que Hephaestus-PL melhorou a configurabilidade e flexibilidade quando comparado à evolução anterior de Hephaestus. A arquitetura de Hephaestus-PL proposta e a abordagem de gerenciamento de variabilidade utilizadas em Hephaestus-PL endereçam a configurabilidade e flexibilidade no desenvolvimento de ferramentas de derivação de LPS. Assim, utilizamos uma abordagem transformacional com operações de metaprogramação para estender os pontos de variação do produto base da instância de Hephaestus-PL.

Em Hephaestus-PL algum grau de modularidade foi obtido pelo mapeamento de configurações de produto de Hephaestus-PL para transformações de metaprogramação onde a variabilidade de Hephaestus-PL relacionada à configuração dos ativos é tratada dinamicamente e automaticamente na geração de uma variante Hephaestus-PL. Isso é permitido pelo processo transformacional de geração de produtos em Hephaestus-PL com o suporte das operações de metaprogramação que atendem às necessidades de gerenciar a variabilidade de ativos nos artefatos tipos de dados abertos e funções abertas do módulo base de Hephaestus. Além disso, o processo reativo definido em Hephaestus-PL para introduzir suporte ao gerenciamento de variabilidades em novos ativos contribui para a flexibilidade de Hephaestus-PL.

Embora o nosso estudo concentra-se em uma única ferramenta, acreditamos que o seu projeto e processo reativo suportado podem ser usados para melhorar a configurabilidade e flexibilidade em outras ferramentas de derivação de produtos de LPS. No entanto, um trabalho empírico é necessário para endereçar a ameaça à validade externa. Planejamos também realizar mais estudos empíricos para avaliar a evolução de Hephaestus-PL para tratar a variabilidade em diferentes tipos de ativos.

Como trabalho futuro, nós propomos a definição de *design rules* que representam um mecanismo que permitirá a redução do tamanho das estruturas de metadados dos ativos em Hephaestus-PL. Usando um processo de inferência nos módulos do ativo, especialmente nos módulos que definem os tipos de dados e as transformações do ativo, seria possível extrair informações atualmente contidas nas estruturas de metadados do ativo e usadas para estender os pontos de variabilidade de um produto base de Hephaestus. Nesse caso, poderia-se reduzir o tamanho das estruturas de metadados dos ativos. Outra solução intermediária que traria uma boa redução das estruturas de metadados de ativos seria trabalhar com duas informações, a *sigla* e o *nome* do ativo, e derivar a maioria das outras informações a partir delas para estender os pontos de variação de Hephaestus-PL.

Capítulo 5

Conclusão

Neste trabalho discutimos a proposta de tratar as ferramentas de derivação de produtos, que dão suporte à Engenharia de Aplicação, como linhas de produtos visando promover o suporte à configurabilidade e flexibilidade nas mesmas. Estas propriedades são atendidas pelo conceito de Linha de Produtos de Software (LPS). Configurabilidade remete à possibilidade de gerar diferentes configurações de produtos e, no caso de ferramentas de LPS, representa a possibilidade de gerar produtos com diferentes combinações de ativos (seleção de features válidas segundo o Modelo de Features). Para isso, é necessário mapear as configurações desejáveis definindo o Modelo de Features a partir da análise do domínio da ferramenta e gerenciar a sua variabilidade. Flexibilidade em LPS representa a possibilidade de evoluir e estender a linha de produtos para atender novos requisitos do domínio (abordagem reativa) e, no caso de ferramentas de LPS, representa a sua evolução para suportar novos ativos de linhas de produtos.

Para estudar a nossa proposta utilizamos a ferramenta Hephaestus e suas várias versões como estudo de caso para o desenvolvimento da linha de produtos Hephaestus (Hephaestus-PL). Realizamos a análise de domínio e identificamos as comunalidades e variabilidades que foram representadas no Modelo de Features de Hephaestus-PL. Além disso, definimos um conjunto de transformações para o processo de geração de produtos Hephaestus-PL. Para atender às variabilidades identificadas nas três versões de Hephaestus definimos um conjunto de seis transformações para Hephaestus-PL. Dentre elas, destacamos a transformação *SelectAsset* que, a partir de um produto base de Hephaestus representando a comunalidade da ferramenta, estende esse produto base adicionando o código associado à feature (ativo de LPS) selecionado. Essas transformações relacionadas às expressões de features no Conhecimento de Configuração de Hephaestus-PL permitem a geração de produtos com qualquer combinação dos ativos casos de uso, requisitos, código fonte e processos de negócio representados no Modelo de Features de Hephaestus-PL. Além disso, essas transformações definidas como transformações de alto nível são capazes de atender a novos ativos na evolução de Hephaestus-PL desde que esses novos ativos apresentem variabilidade similar no código de Hephaestus conforme os ativos já suportados por Hephaestus-PL.

Cada transformação de alto nível é implementada por um conjunto de operações de metaprogramação que representam transformações de baixo nível pois modificam o código de um módulo Haskell (produto base) incorporando trechos de código das features selecionadas durante o processo de geração de instâncias de produtos Hephaestus-PL. Diferentes

transformações de alto nível compartilham a mesma transformação de baixo nível, porém com argumentos diferentes. Os pontos de extensão do produto base que receberão código adicional da feature selecionada estão espalhados e são identificados por parâmetros cujos valores são armazenados internamente à Hephaestus-PL (estrutura de metadados).

Um ponto crítico do processo de gerenciamento de variabilidades neste contexto é definir a técnica mais adequada para o gerenciamento de variabilidades da ferramenta. No caso de Hephaestus-PL o estudo comparativo apresentado no Capítulo 3 mostrou que, considerando o contexto de Hephaestus implementado em linguagem funcional Haskell, nenhuma técnica atenderia na completude as necessidades de Hephaestus-PL mapeadas nas métricas do modelo GQM proposto. Entretanto, as técnicas mais adequadas dentre as analisadas foram CIDE e Transkell de abordagem anotativa e transformacional, respectivamente. Entretanto, outras técnicas como AOP e FOP de abordagem composicional também são aplicáveis ao desenvolvimento de Hephaestus-PL com grau maior de esforço e custo.

A avaliação realizada nos Papers I e II usando um modelo GQM com métricas qualitativas apresenta os seguintes resultados. No Paper I as questões e métricas representam os tipos e as características das variabilidades do projeto Hephaestus-PL o que dá suporta para avaliar a aplicabilidade das técnicas de gerenciamento de variabilidade ao projeto Hephaestus-PL. Além disso, os requisitos de configurabilidade e flexibilidade também estão representados nas métricas definidas. No Paper II, as questões e métricas definidas no modelo GQM foram refinadas considerando a análise detalhada do domínio e a implementação de Hephaesuts-PL trazendo algumas métricas quantitativas como número de módulos e linhas alterados e criados, para medir questões relacionadas à configurabilidade e flexibilidade.

O desenvolvimento de Hephaestus-PL apresenta características importantes como o uso de *bootstrapping*, melhoria no suporte à configurabilidade e flexibilidade de ativos, o uso do mesmo processo de derivação de produtos de Hephaestus e o uso de metaprogramação para o gerenciamento de variabilidades. Hephaestus-PL foi desenvolvido a partir da extração das variantes de Hephaestus e representa um kernel que permite a geração de instâncias de produtos Hephaestus bem como a geração do próprio Hephaestus, considerado também um ativo. A configurabilidade em Hephaestus-PL permite a geração de qualquer combinação válida de ativos a partir do Modelo de Features onde os ativos estão mapeados como *or-features*. A flexibilidade é afetada pelos fatores acoplamento e separação de interesses. Baixo acoplamento e alta separação de interesses promovem a flexibilidade por facilitarem a remoção ou inclusão de funcionalidades. A flexibilidade de Hephaestus-PL é proporcionada pela sua arquitetura e módulos com baixo acoplamento e, ainda, pela definição de um processo reativo para evolução de Hephaestus-PL elaborado neste trabalho.

Hephaestus-PL utiliza o mesmo processo de derivação de produtos de Hephaestus onde os artefatos Modelo de Features, Conhecimento da Configuração e Configuração de Produto são entradas para um processo denominado *build* cujo resultado é uma instância da ferramenta Hephaestus. O uso da técnica de metaprogramação em linguagem Haskell permitiu a configurabilidade automática de ativos em ferramentas Hephaestus resolvendo a variabilidade de *or-features* em tempo de execução na geração de produtos Hephaestus.

No Paper II quando avaliamos os resultados das métricas definidas no modelo GQM, observamos que Hephaestus-PL apresenta vantagens sobre Hephaestus. Como todo de-

senvolvimento de linha de produtos, Hephaestus-PL gerou um esforço maior no seu desenvolvimento inicial mas a sua infraestrutura criada contribui para um esforço menor relacionado às alterações para atender a novos ativos quando comparado com o esforço necessário para inserir novos ativos em Hephaestus. Além disso, em Hephaestus-PL definimos como homogêneas e localizadas as alterações para atender novos ativos pois referem-se apenas a definição de informações de metadados do ativo em um único módulo. Assim sendo, em Hephaestus-PL existe uma maior possibilidade de automação dessas atividades demandadas pela evolução da LPS para atender novos ativos. Uma proposta de automação nesse sentido é definir *design rules* a serem seguidas na implementação dos novos ativos e definir um processo de inferência sobre esses módulos do novo ativo para recuperar as informações usadas pelas operações de metaprogramação. Em contrapartida, em Hephaestus as mudanças para atender novos ativos são heterogêneas, espalhadas e, portanto, com menor possibilidade de automação. São heterogêneas pois referem-se a diferentes tipos de mudanças como inserção de *fields* em tipos de dados, definição de novas instâncias de classe, definição de novas funções e inserção de sentenças em funções. São espalhadas porque as mudanças ocorrem em cinco módulos de Hephaestus. Outra vantagem observada em Hephaestus-PL trata-se do suporte à configurabilidade que ocorre de forma natural (por ser uma LPS) e automática a partir de uma configuração de features do Hephaestus-PL's FM.

A avaliação realizada mostrou que Hephaestus-PL apresenta uma melhor configurabilidade e flexibilidade quando comparado às versões de Hephaestus. Assim, os resultados obtidos são indícios da utilidade da abordagem transformacional para o gerenciamento de variabilidades em Hephaestus-PL e da filosofia de tratar diferentes versões de Hephaestus como uma Linha de Produtos de Software. Acreditamos que tais ideias possam ser utilizadas no desenvolvimento de outras ferramentas para LPS.

5.1 Trabalhos Relacionados

Diferentes técnicas de projeto e implementação de SPL têm sido comparadas (Anastasopoulos e Gacek, 2001; Svahnberg et al., 2005; Lopez-Herrejon et al., 2005). No entanto, isso não têm sido realizado no contexto de ferramentas de desenvolvimento de SPL, como fizemos neste trabalho. O trabalho anterior (Batory et al., 2003) considerou que uma técnica composicional, a programação orientada a aspectos (AOP) implementada em AHEAD, poderia *bootstrap* a ferramenta em si, portanto, potencialmente levando a uma família de ferramentas AHEAD. No entanto, isso não tem sido explorado para tratar a variabilidade em diferentes artefatos, como proposto aqui. Por outro lado, CIDE e CIDE+, abordagens anotativas, podem tratar variabilidade em diferentes artefatos. No entanto, diferentemente do nosso trabalho, nenhuma das abordagens mencionados anteriormente trata a combinação de diferentes artefatos para atender *or-features*.

Da mesma forma a este trabalho, Apel et al. (Apel et al., 2009b) compararam uma abordagem anotativa em relação à composicional visando a modularização de features. Segundo seu estudo, cada uma traz benefícios relativos e concluíram que a combinação das abordagens é mais adequada para modularização de features. A abordagem sinérgica é semelhante a conclusão do nosso estudo apresentado no Capítulo 3, mas nós consideramos outras abordagens na avaliação (transformacional e polimorfismo paramétrico) e concluímos que a abordagem anotativa é melhor se combinada com a transformacional. Além

de Hephaestus, existem outras ferramentas de derivação de produtos SPL (por exemplo, pure::variants, GenARch e Gears), mas, como dito em outros trabalhos (Rabiser et al., 2010) a maioria dessas ferramentas carece de flexibilidade e adaptabilidade.

Transkell (Xavier e Borba, 2010) é uma técnica de abordagem transformacional diferente da técnica metaprogramação utilizada no nosso trabalho e que nos permitiu implementar as transformações na própria linguagem Haskell. Transkell é uma linguagem de domínio específico (DSL) desenvolvida para extrair e modularizar as features representadas pelas transformações de Hephaestus enquanto no nosso trabalho focamos nas variabilidades e features representadas pelos ativos de Hephaestus. Portanto, para atender às features e aos requisitos de variabilidade mapeados no Modelo de Features de Hephaestus-PL precisaríamos desenvolver regras de transformação de código específicas e reescrever os módulos com variabilidade de Hephastus-PL usando a linguagem Transkell. Além disso, para utilizar Transkell seria necessário utilizar o ambiente Stratego/XT (Visser, 2003).

A abordagem *DOPLER (Decision-Oriented Product Line Engineering for effective Reuse)* é um pacote de ferramentas para LPS desenvolvido como uma linha de produtos. *DOPLER* é composta de uma linguagem de modelagem de variabilidades orientada a decisão *DoplerVML* e de uma *meta-ferramenta DOPLER*. *DoplerVML* é uma linguagem de modelagem de variabilidades para definir linhas de produtos baseada em modelos de decisão com ênfase na derivação de produtos. O espaço do problema é modelado usando *modelos de decisão* e o espaço da solução usando *modelos de assets* que representam tipos arbitrários de ativos reusáveis. As decisões e ativos são conectados através de *condições de inclusão* que definem a rastreabilidade entre o espaço do problema e da solução. Diferente do modelo de features que descreve todas as features disponíveis, o modelo de decisão do *DoplerVML* não documenta a análise de domínio da linha de produtos e não necessariamente descreve todas as features disponíveis. *DoplerVML* enfatiza a derivação de produtos usando modelos de decisão que especificam as customizações disponíveis (produtos) a partir da perspectiva do usuário.

Além disso, é usada a ferramenta *ConfigurationWizard* que faz parte do pacote *DOPLER* para o processo de derivação de produtos. Após a escolha de uma decisão pelo usuário, as condições de inclusão são automaticamente avaliadas com as dependências dos ativos para garantir configurações válidas. Todos os ativos cuja condição de inclusão for validada com sucesso serão incluídos no produto final. *DoplerVML* oferece flexibilidade na definição do *modelo de assets* pois permite definir diferentes ativos de domínios específicos. A *meta-ferramenta DOPLER* permite a definição e execução de modelos de variabilidade que podem ser especializados para domínios específicos. Dessa forma, atende a flexibilidade e adaptabilidade mas não suporta a configurabilidade de uma combinação de ativos.

Outros trabalhos discutem o uso da abordagem de linha de produtos para o desenvolvimento de ferramentas de LPS (Grünbacher et al., 2008). Entretanto, esses trabalhos não apresentam detalhes do processo de engenharia de domínio e aplicação utilizados. Além disso, o requisito de configurabilidade em ferramentas de LPS para permitir a geração de produtos com qualquer combinação de artefatos de domínios, não foi explicitamente abordada em trabalhos anteriores.

(Batory et al., 2003) realiza o bootstrapping de AHEAD com AHEAD usando uma linguagem de programação e paradigma diferentes do nosso trabalho, além de não abordar a customização com diferentes artefatos e não oferecer suporte à *or-features*.

Alguns outros estudos comparativos aplicados à ferramentas de derivação de produtos foram realizados (Torres et al., 2010b,a). Em (Torres et al., 2010b) seis ferramentas de derivação de produtos, entre elas Hephaestus, foram analisadas no contexto de evolução da linha de produtos MobileMedia (Figueiredo et al., 2008) para avaliar a modularidade, complexidade e estabilidade no artefato CK da derivação de produtos durante a evolução da linha de produtos. A avaliação mostrou que as ferramentas *GenArch*, *pure::variants* and *Hephaestus* que utilizam um CK modularizado oferecem benefícios para a modularização e estabilidade da LPS, o que consequentemente é favorável à flexibilidade da ferramenta. Em (Torres et al., 2010a) foi realizado um estudo comparativo entre ferramentas de desenvolvimento de LPS que suportam derivação de produtos baseadas em abordagens DDM e DSOA, onde um critério avaliado foi a flexibilidade e extensibilidade das ferramentas para suportar a adição de novas funcionalidades visando integrar facilmente novos módulos e extensões em cada ferramenta.

5.2 Trabalhos Futuros

Como trabalhos futuros, propomos a realização de estudos empíricos para melhor avaliar a técnica de metaprogramação em Hephaestus-PL usada para tratar variabilidades em diferentes ativos durante a evolução de Hephaestus-PL. Esta técnica foi utilizada como abordagem transformacional para o gerenciamento da variabilidade de Hephaestus-PL, uma ferramenta de derivação de linha de produtos, buscando promover o aumento da configurabilidade e flexibilidade de ferramentas de LPS.

Além disso, propomos um estudo em Hephaestus-PL para avaliar a possibilidade de reduzir o tamanho da estrutura de dados que define os metadados utilizados pelas transformações de código (operações de metaprogramação) no produto base de Hephaestus-PL. Nesse sentido, propomos a definição de *design rules* que permitam um mecanismo de inferência nos módulos dos ativos, principalmente os módulos que definem os tipos de dados e as transformações do ativo, extraindo assim informações para as operações de metaprogramação que estendem os pontos de variabilidade do produto base de Hephaestus-PL.

Referências

Vander Alves, Pedro Matos, Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, e Geber Ramalho. Extracting and evolving code in product lines with aspect-oriented programming. *T. Aspect-Oriented Software Development*, 4:117–142, 2007.

Michalis Anastasopoulos e Cristina Gacek. Implementing product line variabilities. In *Symposium on Software Reusability*. ACM Press, May 2001.

Michał Antkiewicz e Krzysztof Czarnecki. Featureplugin: Feature modeling plug-in for eclipse. In *The 2004 OOPSLA Workshop on Eclipse Technology eXchange - Eclipse '04*, pages 67 – 72, Vancouver, British Columbia, Canada, 2004. ACM Press, ACM Press.

Sven Apel, Christian Kästner, e Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *ICSE*, pages 221–231, 2009a.

Sven Apel et al. Feature (de)composition in functional programming. In *SC '09*, pages 9–26, 2009b.

Victor R. Basili, Gianluigi Caldiera, e H. Dieter Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

Don S. Batory. Feature-oriented programming and the ahead tool suite. In *ICSE*, pages 702–703, 2004.

Don S. Batory, Richard Cardone, e Yannis Smaragdakis. Object-oriented frameworks and product lines. In *SPLC*, pages 227–248, 2000.

Don S. Batory, Jacob Neal Sarvela, e Axel Rauschmayer. Scaling step-wise refinement. In *ICSE*, pages 187–197, 2003.

Don S. Batory, Jacob Neal Sarvela, e Axel Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

Danilo Beuche. Modeling and building software product lines with pure:: Variants. In *SPLC*, page 358, 2008.

R. Bonifácio. *Modeling Software Product Line Variability in Use Case Scenarios - An Approach Based on Crosscutting Mechanisms*. PhD thesis, Universidade Federal de Pernambuco, fevereiro 2010.

Rodrigo Bonifácio e Paulo Borba. Modeling scenario variability as crosscutting mechanisms. In *AOSD '09*, pages 125–136. ACM, 2009.

Rodrigo Bonifácio, Leopoldo Teixeira, e Paulo Borba. Hephaestus: A tool for managing SPL variabilities. In *SBCARS Tools Session*, 2009. A draft copy is available at: http://bit.ly/notgBg.

Virgilio Borges, Rógel Garcia, e Marco Tulio Valente. Cide +: A tool for semi-automatic extraction of software product lines using coloring code (in portuguese). In *SBSOFT 2010 Tools Session*, pages 73–78, 2010.

Elder Cirilo, Ingrid Nunes, Uirá Kulesza, e Carlos José Pereira de Lucena. Automating the product derivation process of multi-agent systems product lines. *Journal of Systems and Software*, 85(2):258–276, 2012.

Paul Clements e Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

Marcus Vinicius Couto, Marco Tulio Valente, e Eduardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In *CSMR*, pages 191–200, 2011.

Krzysztof Czarnecki e Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

Sybren Deelstra, Marco Sinnema, e Jan Bosch. Product derivation in software product families: a case study. *JSS*, 74(2):173–194, 2005.

Deepak Dhungana, Paul Grünbacher, e Rick Rabiser. The dopler meta-tool for decision-oriented variability modeling: a multiple case study. *Autom. Softw. Eng.*, 18(1):77–114, 2011.

Oscar Díaz, Salvador Trujillo, e Felipe I. Anfurrutia. Supporting production strategies as refinements of the production process. In *SPLC*, pages 210–221, 2005.

Felype Ferreira, Lais Neves, Michelle Silva, e Borba Paulo. Target: a model based product line testing tool. In *CBSoft 2010 Tools Session*, pages 1–4, 2010.

Eduardo Figueiredo, Nélio Cacho, Cláudio Sant'Anna, Mario Monteiro, Uirá Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Cutigi Ferrari, Safoora Shakil Khan, Fernando Castor Filho, e Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE*, pages 261–270, 2008.

Geam Filgueira. Crossmda-spl: Uma abordagem para gerência de variações baseada em modelos e aspectos, 2009.

Martin L. Griss. Implementing product-line features with component reuse. In *ICSR*, pages 137–152, London, UK, 2000.

Paul Grünbacher, Rick Rabiser, e Deepak Dhungana. Product line tools are product lines too: Lessons learned from developing a tool suite. In *ASE*, pages 351–354, 2008.

M. Jones. Functional programming with overloading and higher-order polymorphism. *Advanced Functional Programming*, pages 97–136, 1995.

Edison Kicho Shimabukuro Junior. Um gerador de aplicações configurável, 2006.

Christian Kästner, Sven Apel, e Martin Kuhlemann. Granularity in software product lines. In *ICSE '08*, pages 311–320, 2008.

Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, e Don S. Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *TOOLS (47)*, pages 175–194, 2009.

Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, e Sven Apel. Featureide: A tool framework for feature-oriented software development. In *ICSE*, pages 611–614, 2009.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, e John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, e William G. Griswold. An overview of aspectj. In *ECOOP*, pages 327–353, 2001.

Günter Böckle Klaus Pohl e Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer,, 2005.

Charles Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering.*, pages 282–293, Bilbao, Spain, October 2001.

Charles W. Krueger. Biglever software gears and the 3-tiered spl methodology. In *OOPSLA Companion*, pages 844–845, 2007.

Ralf Lämmel e Klaus Ostermann. Software extension and integration with type classes. In *GPCE '06*, pages 161–170, 2006.

Roberto E. Lopez-Herrejon, Don S. Batory, e William R. Cook. Evaluating support for features in advanced modularization technologies. In *ECOOP*, pages 169–194, 2005.

Idarlan Machado, Rodrigo Bonifácio, Vander Alves, Lucinéia Turnes, e Giselle Machado. Managing variability in business processes: an aspect-oriented approach. In *EA '11*, pages 25–30. ACM, 2011.

Hidehiko Masuhara, Hideaki Tatsuzawa, e Akinori Yonezawa. Aspectual caml: an aspect-oriented functional language. *SIGPLAN Not.*, 40:320–330, September 2005.

Mira Mezini e Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT FSE*, pages 127–136, 2004.

Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulesza, e Paulo Borba. Investigating the safe evolution of software product lines. In *GPCE*, pages 33–42, 2011.

OMG. Software process engineering metamodel (spem), 2008. URL `http://www.omg.org/spec/SPEM/2.0/`.

David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976.

C. A. F. Pereira, R. T. V. Braga, e P. C. Masieiro. Captor-ao: Gerador de aplicações apoiado pela programação orientada a aspectos. In *Anais da Seção de Ferramentas do II Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS'2008)*, pages 1–8, October 2008. ISBN 978-85-7430-796-1.

Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443, 1997.

Rick Rabiser, Paul Grünbacher, e Deepak Dhungana. Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Inf. Softw. Technol.*, 52:324–346, March 2010.

Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, e Nico Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, pages 77–91, 2010a.

Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, e Nico Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, pages 77–91, 2010b.

Marco Sinnema e Sybren Deelstra. Industrial validation of covamof. *Journal of Systems and Software*, 81(4):584–600, 2008.

Eduardo Steiner, Paulo Masiero, e Rodrigo Bonifácio. Managing variability in an unmanned aerial vehicle's simulink model with pure::variants and hephaestus (in portuguese). In *Ibero-American Conference on Software Engineering (CIbSE)*, pages 1–14, 2012.

Mikael Svahnberg, Jilles van Gurp, e Jan Bosch. A taxonomy of variability realization techniques. *Softw., Pract. Exper.*, 35(8):705–754, 2005.

Soe Myat Swe, Hongyu Zhang, e Stan Jarzabek. Xvcl: a tutorial. In *SEKE*, pages 341–349, 2002.

Mário Henrique C. Torres, Uirá Kulesza, Rosana T. V. Braga, Paulo Cesar Masiero, Paulo F. Pires, Flávia C. Delicato, Elder Cirilo, Thaís Vasconcelos Batista, Leopoldo Teixeira, Paulo Borba, e Carlos José Pereira de Lucena. Estudo comparativo de ferramentas de derivação dirigidas por modelos: Resultados preliminares. In *Anais do I Workshop Brasileiro de Desenvolvimento Dirigido por Modelos (WBDDM'2010) - CBSoft'2010*, pages 1–8, 2010a.

Mário Henrique C. Torres, Uirá Kulesza, Matheus Sousa, Thaís Vasconcelos Batista, Leopoldo Teixeira, Paulo Borba, Elder Cirilo, Carlos José Pereira de Lucena, Rosana T. V. Braga, e Paulo Cesar Masiero. Assessment of product derivation tools in the evolution of software product lines: an empirical study. In *FOSD*, pages 10–17, 2010b.

Lucinéia Turnes, Rodrigo Bonifácio, Vander Alves, e Ralf Lämmel. Techniques for developing a product line of product line tools: a comparative study. In *Proceedings of 5th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS 2011) at CBSoft 2011 Brazilian Conference on Software: Theory and Practice*, pages 1–10, 2011.

Michael Vierhauser, Gerald Holl, Rick Rabiser, Paul Grünbacher, Martin Lehofer, e Uwe Stürmer. A deployment infrastructure for product line models and tools. In *SPLC*, pages 287–294, 2011.

Eelco Visser. Syntax definition for language prototyping, 1997.

Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In *LNCS*, volume 3016, pages 216–238. Springer, 2003.

Markus Voelter e Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society.

Philip Wadler. The expression problem. message to java-genericity electronic mailing list, 1998. Available online at http://www.daimi.au.dk/madst/tool/papers/expression.txt.

Meng Wang e Bruno C. d. S. Oliveira. What does aspect-oriented programming mean for functional programmers? In *WGP '09*, pages 37–48. ACM, 2009.

Marcos Augusto Xavier e Paulo Borba. A DSL specification for implementing SPL variability, 2010.