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

Análise Automática do Modelo de Features em Linha de Produtos de Software

Luiz José de Brito

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio

Brasília
2013

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

Coordenador: Prof. Dr. Maristela Terto de Holanda

Banca examinadora composta por:

Prof. Dr. Rodrigo Bonifácio (Orientador) — CIC/UnB
Prof. Dr. Flávio Leonardo Cavalcanti de Moura — CIC/UnB
Prof. Dr. Genaina Nunes Rodrigues — CIC/UnB

CIP — Catalogação Internacional na Publicação

Brito, Luiz José de.

Análise Automática do Modelo de Features em Linha de Produtos de Software / Luiz José de Brito. Brasília : UnB, 2013.
125 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2013.

1. linha de produtos para software, 2. modelo de features,
3. satisfatibilidade

CDU 004.4

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

Análise Automática do Modelo de Features em Linha de Produtos de Software

Luiz José de Brito

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Flávio Leonardo Cavalcanti de Moura
CIC/UnB

Prof. Dr. Genaina Nunes Rodrigues
CIC/UnB

Prof. Dr. Maristela Terto de Holanda
Coordenador do Bacharelado em Ciência da Computação

Brasília, 1 de setembro de 2013

Dedicatória

Deus, meu Pai e minha Mãe, minhas irmãs e irmão, minha Esposa Jaqueline e meu Filho Luiz Henrique.

Agradecimentos

A lista para agradecimentos é grande: meu pai e minha mãe, minhas irmãs e irmão, minha esposa Jaqueline e meu filho Luiz Henrique que me motivaram neste difícil, mas empolgante curso de graduação; amigos do Tribunal de Contas da União pelo incentivo permanente aos estudos; meu orientador - Professor Rodrigo Bonifácio, sempre disponível e otimista neste trabalho de conclusão de curso. E a Deus que sempre me inspirou com a esperança de tentar superar minhas limitações.

Abstract

A abordagem de Linhas de Produtos de *Software* (LPS) se preocupa com o desenvolvimento de aplicações utilizando técnicas sistemáticas de reuso de *software*, capturando características (*features*) comuns e variáveis de de uma família de sistemas pertencentes a um mesmo domínio. As características de uma LPS são comumente representadas utilizando modelos de *features*, que guiam a derivação (automática) dos produtos. Com isso, torna-se necessária a utilização de suporte ferramental não apenas para o processo de derivação de produtos, mas também a própria análise dos modelos de *features*. A ferramenta *Hephaestus* possui funcionalidades para atender a ambos os objetivos, apesar de não suportar todas as operações de análise aplicáveis aos modelos de *features*. Este trabalho contribui com a implementação de novas operações para a ferramenta *Hephaestus*, além de realizar uma análise extensiva da escalabilidade dos modelos de features com diferentes graus de complexidade.

Palavras-chave: linha de produtos para software, modelo de features, satisfatibilidade

Abstract

The approach of Product Lines Software (SPL) is concerned with the development of applications using techniques systematic reuse of software , capturing features common and variable of a family of systems belonging to the same domain. the characteristics of an SPL are commonly represented using models of features, which guide the derivation (automatic) of products. Therefore , it becomes necessary to use supports tooling not only for the process of deriving products, but also own analysis of models of features. The tool Hephaestus has features to suit both goals , despite not supporting all operations analysis applicable to models features. This work contributes with the implementation of new business for the tool Hephaestus , and perform an extensive analysis of scalability features of the model with different degrees of complexity.

Keywords: reuse, software product lines, features

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O aumento vertiginoso na velocidade dos processadores e capacidade de armazenamento levou a uma demanda crescente no desenvolvimento de aplicações mais complexas. Para atender a essa demanda, novas metodologias e técnicas com foco em reuse de *software* foram propostas. Para Krueger [1], o reuso de *software* ocorre quando um sistema é desenvolvido a partir de pedaços de *software* já existente. Na forma mais básica, o reuso é uma técnica utilizada naturalmente pelos programadores— seja com a cópia de trechos de código ou utilizando mecanismos de abstração, composição e herança, a fim de resolver problemas similares. Por outro lado, essas formas de reuso são *ad hoc*, caracterizadas pela falta de sistematização. Isso leva a um baixo grau de reutilização.

Para contornar essse problema, surgiram abordagens de desenvolvimento, como Linhas de Produtos de *Software* (LPS) e *Software Factories*, que têm forte preocupação com reuso. A abordagem de LPS, segundo Clements [2], é um conjunto de sistemas intensivos de *software*, compartilhando características, que é gerenciado para satisfazer as necessidades específicas do mercado. Consiste em uma proposta de desenvolvimento sistemático *para* e *com* reuso, capturando explicitamente as características (*features*) comuns e as variabilidades de sistemas que constituem uma linha de produtos. A principal diferença entre o desenvolvimento tradicional de *software* e a abordagem com LPS está no deslocamento do foco: de um sistema individual para uma linha de produtos. É a transposição (inspiração) das técnicas de engenharia de fabricação de produtos industriais para a produção de *software*. De forma resumida, os objetivos pretendidos pela LPS são [3]: reduzir os custos de desenvolvimento, melhoria de qualidade dos produtos, redução do tempo de entrega dos produtos, redução do esforço de manutenção; e para isso deverá desenvolver técnicas que identifiquem as semelhanças dos produtos e gerenciem as suas variabilidades.

Considerando uma perspectiva histórica, apesar do termo LPS ter entrado em mais evidência no final dos anos 90, ainda na década de 1970 Parnas apresentou o conceito de família de produtos [4]. No artigo *On the Design and Development of Program Families*, Parnas propôs modelos de desenvolvimento para *software*, visando maior produtividade com o agrupamento de características comuns. Já o conceito de linhas de produtos foi efetivamente introduzido no início dos anos 1990, com a descrição da abordagem *Feature-Oriented Domain Analysis*— FODA que contribuiu para estabelecer o conceito de modelo de *features*. *Features* são atributos de um sistema que afetam diretamente o usuário final. A Figura 1.1 exibe um modelo de *features* para o *middleware* Ginga utilizado em conver-

sores, televisores e dispositivos portáteis para o Sistema Brasileiro de Televisão Digital, com funcionalidades para recebimento e processamento do sinal de TV digital, além de prover diversos serviços e funcionalidades que podem ser escolhidas pelo o usuário [5].

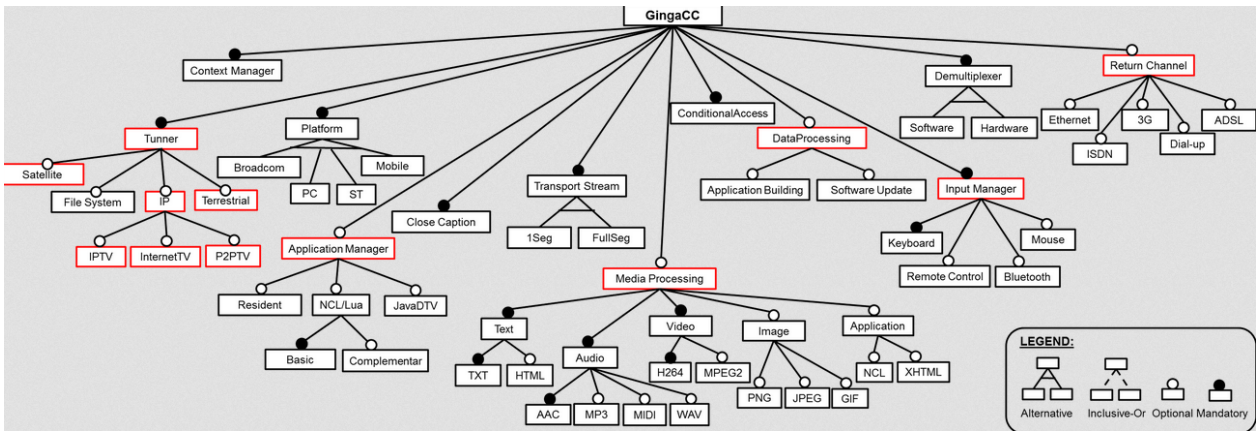


Figura 1.1: Exemplo de modelo de *features* para o *middleware* ginga [5]

Dessa forma, o modelo de *features* se torna essencial não apenas para delimitar o escopo de uma LPS, mas também guiar o processo de derivação de um produto a partir das suas características (*features*) selecionadas. Dada a importância desse artefato, uma série de operações foram propostas para verificar a consistência dos modelos de features automaticamente. A automação em LPS também é essencial no processo de derivação de produtos, o que levou ao desenvolvimento de ferramentas de apoio a essa atividade. Por exemplo, a ferramenta *Hephaestus* suporta operações para raciocinar sobre modelos de LPS (como o modelo de *features*) e suporta a derivação de produtos [6]. Apesar de ser um ferramenta com certa maturidade, e contar com contribuições de diferentes instituições (com UFPE, USP São Carlos e *University of Koblenz-Landau*), uma série de checagens de modelo de *features* consolidadas em um *survey* de Benavides et al [7], não estão implementadas por *Hephaestus*.

1.1 Problema

Conforme mencionado, a ferramenta *Hephastus* não implementa todas as operações de checagem de modelos de features discutidas na literatura [7] e a escalabilidade dessas operações de análise ainda não foram investigadas. Escalabilidade, nesse contexto, refere-se a relação entre o número de *features* e restrições de um modelo de *features* e o tempo de resposta para realizar uma determinada operação.

1.2 Objetivos

Este trabalho tem como objetivo evoluir o suporte e a verificação de modelos de *features* na ferramenta *Hephaestus*, e verificar a escalabilidade das implementações. Mais especificamente:

- Revisar a literatura a fim de identificar quais propriedades de modelos de *features* são importantes para serem verificadas.
- Implementar as funções necessárias para verificar as propriedades relevantes dos modelos de *features*.
- Realizar testes de desempenho, comparando alternativas de implementação e verificando a escalabilidade das implementações.

1.3 Organização dos Capítulos

O segundo e o terceiro capítulos são dedicados ao referencial teórico deste trabalho, apresentando noções sobre a abordagem de Linhas de Produtos de Software, modelos de *features*, operações de análise de modelos de *features*. O quarto capítulo apresenta detalhes das implementações das novas operações adicionadas ao módulo de *features* na ferramenta *Hephaestus*. O quinto capítulo é dedicada à analise dos resultados com testes de desempenho realizados sobre as novas operações. Finalmente, o sexto capítulo apresenta algumas considerações finais e perspectivas de trabalhos futuros.

Capítulo 2

Referencial Teórico

Este capítulo apresenta os conceitos relacionados ao trabalho, como a caracterização das fases de desenvolvimento da LPS: engenharia de domínio e engenharia de aplicação, e a necessidade de suporte ferramental a fim de proporcionar a automatização dessa abordagem sistemática de reuso.

2.1 Linha de Produtos de *Software*

A criação da linha de montagem em série na indústria automobilística por Henry Ford na qual operários especializados trabalhando de forma sequencial, em funções específicas, com ajuda de máquinas [8], obtinham um produto semi-acabado ou acabado; influenciaram as modernas linhas de produção. A produção em série permitiu a diminuição do tempo de entrega do produto e dos custos de fabricação, mas por outro lado reduziu a diversificação dos produtos.

Outro movimento na indústria foi a customização em massa definida como a produção de bens e serviços para atender aos anseios específicos de cada cliente [9]. Novamente a indústria automobilística introduziu um novo conceito - plataformas -, uma mesma base para produção de diferentes tipos de carros. Plataforma é qualquer base tecnológica na qual outras tecnologias ou processos são construídos [10]. A base da customização em massa é a flexibilidade para obtenção de produtos diversos. A combinação da customização em massa com plataformas permitiu o reuso da base comum com os desejos dos consumidores.

Esses conceitos de linhas de produção e customização em massa foram aproveitados na linha de produtos de *software*(LPS). Na definição de Pohl et al. [3], Linha de Produtos de *Software* é um paradigma para desenvolvimento de sistemas usando plataformas e customização em massa.

Os principais objetivos da LPS são identificar as semelhanças e gerenciar as variações a fim de obter melhorias na qualidade, tempo e esforço de desenvolvimento, além da redução de custos e de complexidades no desenvolvimento, pretendendo facilitar a manutenção da linha de produtos.

A redução de custos decorrre principalmente do reuso da plataforma que é comum para a família de produtos gerados. Há de se considerar, entretanto que deve ocorrer um maior investimento para o desenvolvimento da plataforma. A figura 2.1 mostra a comparação dos investimentos iniciais em desenvolvimentos de sistemas com e sem LPS [11]. Apesar

do alto custo inicial com LPS, aposta-se que a partir do cruzamento dos gráficos (com a produção de três produtos em média, e baseado em estudos existentes [2]), a plataforma estará pronta para ser reutilizada em sistemas futuros.

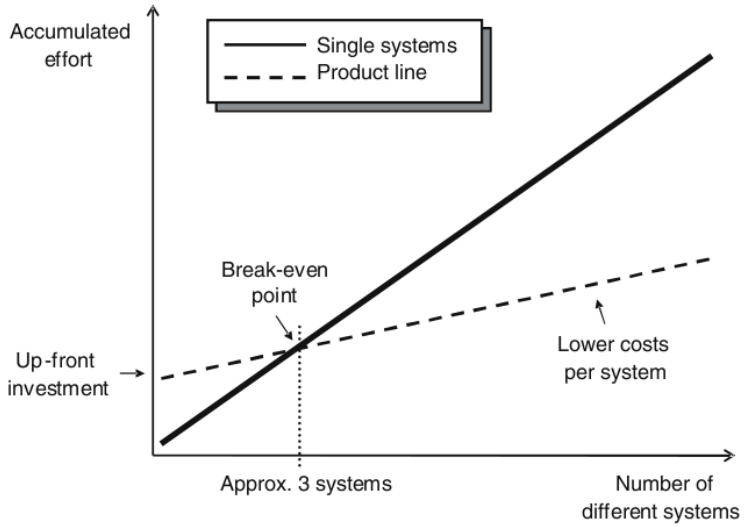


Figura 2.1: Comparação LPS e desenvolvimento de *software* tradicional [11]

A qualidade dos produtos com a LPS é gradativamente alcançada com a reutilização dos artefatos (componentes, casos de uso, diagramas de classes, documentos de projeto) produzidos nas fases de desenvolvimento, uma vez que os artefatos são utilizados e revisados nos diversos sistemas. Em caso de erros (*bugs*), estes são corrigidas e propagadas para os produtos derivados.

A redução do tempo de desenvolvimento é uma das vantagens mais esperadas. No início do desenvolvimento da plataforma, o tempo investido é até maior que das metodologias de desenvolvimento tradicional, entretanto tal situação é superada com reutilização dos artefatos nos sistemas desenvolvidos com as plataformas.

A facilidade na manutenção decorre do fato de que se um artefato é modificado em virtude de uma correção ou atualização, tal mudança deverá ser propagada para os produtos derivados. Tal medida além de garantir o correto funcionamento do artefato, também permite a evolução dos produtos, e por conseguinte dos sistemas produzidos.

Outro objetivo pretendido com a linha de produtos de *software* é prover o desenvolvedor de estimativas de custos dos sistemas desenvolvidos, considerando tanto a plataforma quanto os artefatos derivados, em tese consolidados em face da utilização no mercado.

Há de se destacar os benefícios para os próprios consumidores com a padronização das interfaces de utilização, procedimentos de instalação e operacionalização dos produtos, aspectos importantes para facilitar o uso e a aprendizagem. Para os consumidores que almejam produtos específicos (customizados), os benefícios são percebidos com a redução dos custos e tempo de desenvolvimento.

Por outro lado, a LPS traz alguns riscos. Por se tratar de uma nova estratégia técnica, requer uma eficiente gestão organizacional capaz de adotar boas práticas de técnicas de engenharia a fim de explorar as similaridades dos produtos e de monitorar os esforços de de desenvolvimento.

O projeto de implantação de uma LPS por ser conduzido de duas formas [12]: repentino (completo) - é um processo mais radical e muitas vezes inviável em termos econômicos. Nesse caso, a empresa interrompe o desenvolvimento de novos produtos, realocando os recursos para o desenvolvimento de um repositório de artefatos (arquitetura, componentes, definição de processo); gradual (incremental) - uma abordagem mais racional. Como a interrupção das atividades de produção pode ser fatal, a empresa permanece criando os produtos tradicionais (*single sistems*), e durante a fase de cada projeto são feitas contribuições para a estrutura da linha de produto. Essas contribuições consistem no desenvolvimento de ativos (*assets*) para o repositório de uma linha de produtos.

As dificuldades de implantação de uma LPS são diversas: abordagem inadequada - apesar do foco da LPS está no ganho de produtividade, se os produtos gerados não possuírem similaridades não garantirão a viabilidade do projeto; interação insuficiente entre as equipes - a LPS é um processo que necessita da colaboração entre equipes de marketing, engenheiro de domínio e de aplicação.

2.2 Dois Processos de Desenvolvimento da LPS

Na classificação adotada por Pohl [3], a linha de produtos de *software* pode ser separada em dois processos (Figura 2.2): engenharia de domínio e engenharia de aplicação. Na engenharia de domínio, a plataforma é construída. É a base para o desenvolvimento de produtos individuais. Nesse processo são identificados os aspectos comuns e a variabilidade da linha de produtos [15]. Na engenharia de aplicação o foco é a variabilidade da LPS e na construção de aplicações específicas para os consumidores.

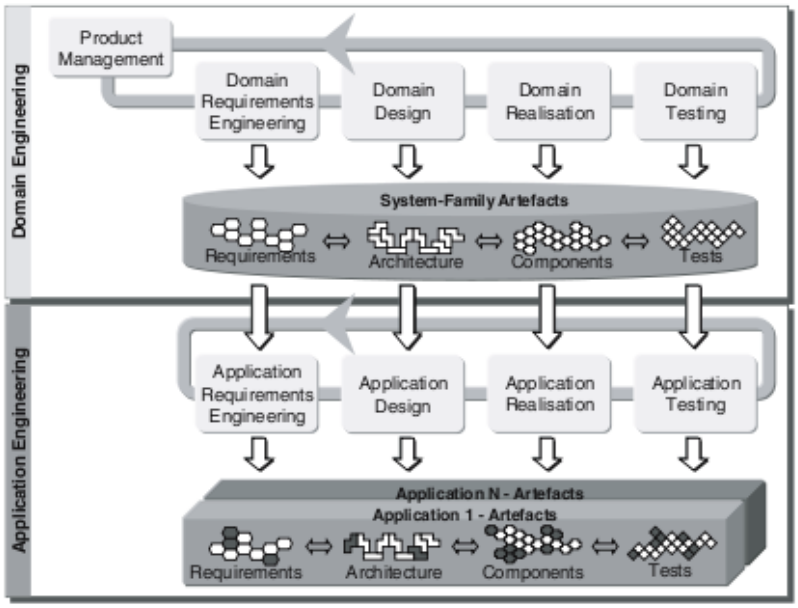


Figura 2.2: As duas fases do modelo de linha de produtos de *software* [11]

2.2.1 Engenharia de Domínio

Na engenharia de domínio (desenvolvimento para reuso) a plataforma é definida como uma coleção de artefatos que serão reutilizados de modo sistemático e ordenada para a construção das aplicações. É composto por cinco sub-processos: gerência de produtos, engenharia de requisitos, projeto de domínio, implementação do domínio e teste de domínio.

- Gerência de Produto - trata do escopo comercial da linha de produtos. O escopo determina as *features* comuns e variáveis da linha de produtos. A entrada de informação deste sub-processo são os objetivos do negócio. A saída da gerência de produtos é o roteiro com as *features* dos futuros produtos.
- Engenharia de requisitos de domínio - abrange todas as atividades de identificação e documentação dos requisitos comuns e variáveis da linha de produtos com participação dos clientes (*stakeholders*). A entrada de informação deste sub-processo consiste do roteiro de produtos. A saída são os modelos de baseados nos requisitos, em especial o modelo de variabilidade da linha de produtos.
- Projeto de Domínio - abrange todas as atividades para definir a arquitetura da linha de produtos. A entrada consiste nos requisitos de domínio e modelo de variabilidade oriundos da engenharia de requisitos de domínio. A saída deste sub-processo é a arquitetura, além de refinar o modelo de varibilidades.
- Implementação do domínio - trata do projeto e implementação das *assets* - componentes reutilizáveis de software. Esses componentes incorporam configurações para implementação da variabilidade na linha de produtos. A entrada consiste da referência da arquitetura incluíndo a lista de artefatos a serem desenvolvidos neste sub-processo. A saída abrange o projeto e a implementação das *assets*.
- Teste de domínio - trata da validação e verificação dos componentes reutilizáveis. Os componentes são confrontados com os requisitos. A entrada consiste de requisitos de domínio, referência de arquitetura, projetos de componentes e interfaces, além dos componentes reutilizáveis. A saída informa os resultados dos testes, inclusive o desempenho.

2.2.2 Engenharia de aplicação

Na engenharia de aplicação (desenvolvimento com reuso), os produtos são derivados a partir da plataforma definida na engenharia de domínio. Sendo composto por quatro sub-processos: engenharia de requisitos de aplicação, projeto de aplicação, implementação de aplicação e teste da aplicação.

- Engenharia de aplicação de requisitos - consolida a especificação de um produto específico, considerando os requisitos dos clientes. A entrada compreende os requisitos de domínio e o roteiro do produto com as principais *features* das implementações.
- Projeto de aplicação - consiste na produção da arquitetura da implementação. Aqui seleciona e configura as partes da arquitetura, incorporando adaptações. É derivada da arquitetura de referência (fase da engenharia de domínio). Entrada consiste,

como já foi dito, da arquitetura de referência e da especificação dos requisitos da aplicação. A saída compreende a arquitetura da aplicação.

- Implementação da aplicação - é a implementação do produto. Consiste na criação da aplicação. A principal preocupação é a seleção e configuração dos componentes de software reutilizáveis.
- Teste de aplicação - compreende atividades para validar e verificar a aplicação em função do que foi solicitado nas especificações. Entrada compreende todos os tipos de artefatos da aplicação. A saída consiste nos relatórios de testes de desempenho.

2.3 Gerência de Variações

A variabilidade dos produtos da linha de produtos modela o desenvolvimento da linha de produto. A LPS pretende gerar uma quantidade significativa de produtos a partir de uma plataforma comum, logo deverá dispor de instrumentos para gerenciar o escopo. Esses produtos devem satisfazer às demandas dos clientes. A variabilidade é capturada [3] nos requisitos, arquitetura, componentes, testes. Cada sub-processo da engenharia de aplicação se liga à variabilidade introduzida no sub-processo correspondente da engenharia de domínio.

Pohl [3] propõe as seguintes questões para identificar a variabilidade:

- *O que varia?* - a resposta dessa questão identifica itens e propriedades do mundo real (variabilidade subjetiva).
- *Por que varia?* - identifica as variabilidades visíveis e ocultas . Outros aspectos devem ser considerados que são as necessidades das partes interessadas (*stakeholders*), normais legais.
- *Como varia?* - trata diretamente da variabilidade subjetiva (propriedades do mundo real) de modo a instanciá-la (variabilidade objetiva).
- *Para quem a variabilidade será documentada?* - considerar o leitor do documento de variabilidades.

2.4 Modelo de *Features*

Os produtos em uma LPS são diferenciados considerando suas características. As linhas de produtos são especificadas usando o modelo de *features*. Esse termo foi utilizado pela primeira vez por Kang [13] em 1990 no *Feature-Oriented Domain Anaysis*(FODA) que buscou representar características e suas relações dentro de um domínio de uma família de sistemas. O principal objetivo da modelagem de *features* [18] é identificar os aspectos comuns e as diferenças entre todos os produtos da LPS. O resultado desse processo é uma compacta representação dos possíveis produtos de uma LPS. Dependendo do estágio de desenvolvimento, uma *feature* pode se referir a um requisito, componente de arquitetura, trechos de código.

O modelo de *features* é representado por um conjunto de características de um domínio que estão dispostas hierarquicamente exibindo relacionamentos e restrições. O diagrama

de *features* é uma árvore, em que a raiz representa um conceito [19] e as folhas são as *features*.O diagrama de *features* oferece uma notação simples e intuitiva para representar pontos de variação. Na Figura 2.3 é apresentado um modelo de *features* para um sistema integrado de segurança residencial.

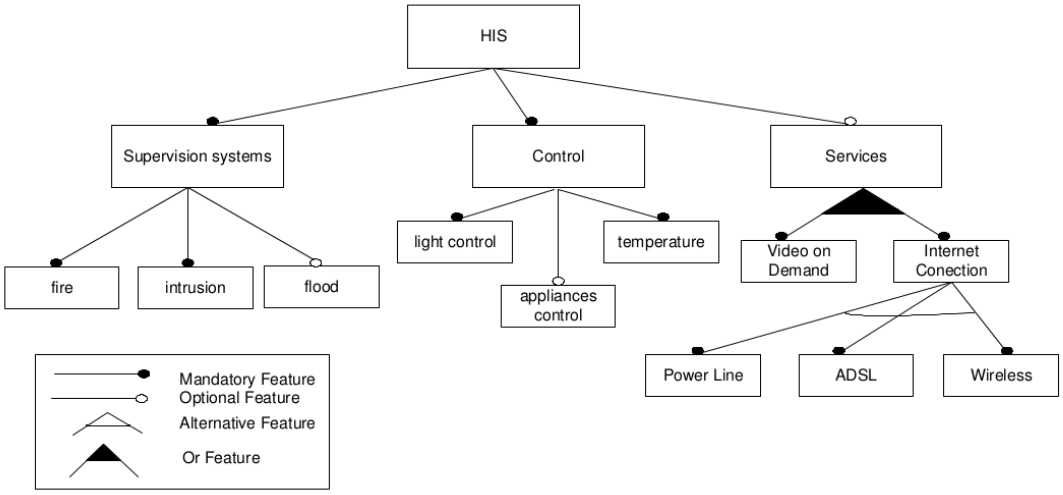


Figura 2.3: Modelo de *features* para domínio de sistema de segurança residencial [18]

O exemplo para um modelo de *features* de um domínio de segurança residencial (Figura 2.3) ilustra a notação tipicamente utilizada para representar os relacionamentos entre as *features* - obrigatória, opcional, alternativa e *or*, cujas definições são as seguintes:

- Obrigatória - uma *feature* filha tem um relacionamento obrigatório com o pai, assim caso o pai apareça num produto, a *feature* filha também aparece. No exemplo do Sistema de Segurança, obrigatoriamente serão criadas as *features*: *HIS* - pai, sendo *Supervision systems* e *Control* - filhas.
- Opcional - a *feature* filha tem uma relação opcional sempre que puder ser incluída facultativamente. No exemplo dado, a *feature flood*(inundação) poderá ser incluída ou não num modelo selecionado de *features*.
- Alternativa - é uma relação exclusiva (xor). Sendo uma *feature* selecionada, as demais serão excluídas. No exemplo, é possível optar somente por um modo de conexão para internet (ADSL, *wireless*).
- Ou inclusivo (*Or*) - são características que podem ser incluídas adicionalmente aos produtos. No sistema de segurança, é facultativo escolher pela filmagem da residência ou utilizando a conexão da internet.

O modelo de *feature* ainda possui relacionamentos transversais entre diferentes subárvores: requerida - quando uma *feature*, sempre que selecionada, requer presença de outra *feature*; exclusão - é o oposto do relacionamento anterior, em que a presença de uma *feature* impede a seleção de outra *feature*.

Além dessas características, outros aspectos [18] também estão disponíveis para análise:

- Relação de cardinalidade entre as *features* - sequência de um intervalo $[n..m]$ que determinará o número de instâncias de uma *feature*;
- Grupo de cardinalidade - é o intervalo $< n..m >$ que limita o número de *features* filhas quando uma determinada *feature* pai for selecionada;
- Atributos - qualquer característica da *feature* que pode ser mensurada. No exemplo da Figura 2.4, as diferentes conexões de internet, representadas pelas *features* - *Power Line, ADSL, Wireless*, possui valores de velocidade e preço;
- Domínio dos atributos - é o espaço de valores permitidos para os atributos (inteiros, booleanos);
- Características extra-funcionais - relação entre os atributos de uma mesma *feature*. No exemplo do sistema extendido de segurança residencial (Figura 2.4), caso seja selecionado o atributo com largura de banda=10M, a disponibilidade de acesso será 10 computadores ligados ao sistema.

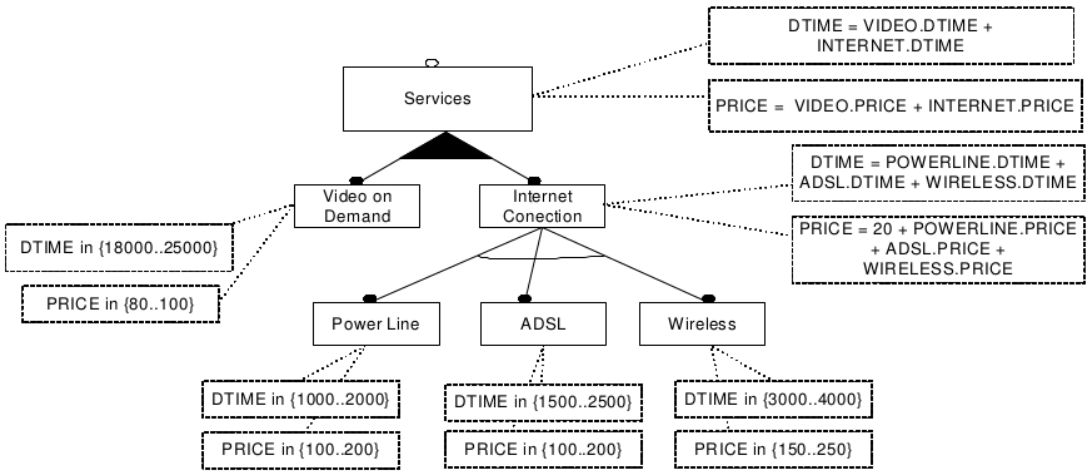


Figura 2.4: Extensão para o modelo de *feature* do sistema de segurança residencial [18]

2.5 Suporte Ferramental

O modelo de *features* apresentado na seção anterior guia a geração automática de produtos, permitindo o relacionamento entre as *features* e os *assets*. Essa integração decorre das fases de desenvolvimento de domínio e aplicação.

Krueger [1] aponta dificuldades na integração dos processos de desenvolvimento de linhas de produtos de *software* devido à segregação das fases - engenharia de domínio e engenharia de aplicação -, decorrente da necessidade de uma configuração específica

para cada produto tendo um contexto isolado para as *core assets* provocando interesses conflitantes entre as equipes de desenvolvimento -*domínio versus aplicação*.

A sistematização da contrução desses produtos será alcançada com a composição automáticas dos componentes. Nas primeiras gerações dos métodos de desenvolvimento [1] de LPS enfatizava a dicotomia de atividades: havia engenheiros focados no domínio do problema, que criavam componentes centrais focados na reutilização; enquanto um outro grupo se utilizava desses componentes para desenvolver os produtos. Nessa separação de atividades surgiam dificuldades de composição e configuração de componentes, bem como apareciam disputas entre as equipes (domínio e aplicação).

Em substituição dessas técnicas separadas, surgiram os configuradores automáticos. Na Figura 2.5 é mostrado um configurador que recebe duas entradas partes centrais (*cores assets*) e modelos de produtos (*product models*), criando automaticamente instâncias de produtos.

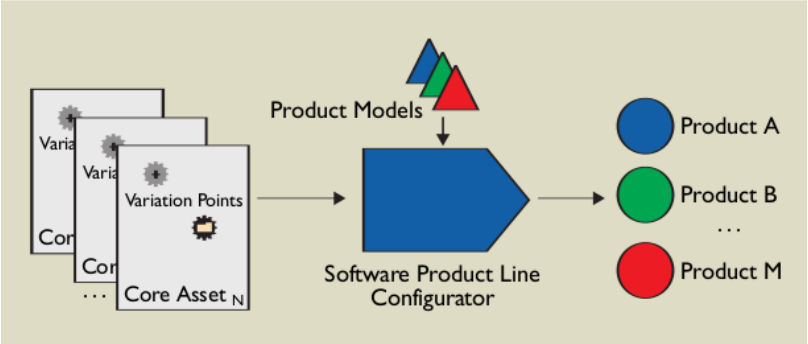


Figura 2.5: Configurador de LPS [1]

Um configurador automático para LPS deverá gerar produtos para LPS de forma segura, verificando as propriedades dos produtos [14] para constatar a consistência ou presença de erros (inspeção manual seria impraticável para um grande número de *features*). Existem diversos configuradores automáticos para modelo de *features* como a ferramenta *Hephaestus*, descrita na próxima seção.

2.6 *Hephaestus*

Hephaestus envolve um conjunto de bibliotecas e ferramentas que permitem avaliar os modelos LPS e auxiliar no processo de derivação de produtos (suportando a engenharia de aplicação). O *Hephaestus* permite a seleção (Figura 2.6) de modelos de entrada, derivando modelos de caso de uso, mapeamento de nomes e códigos fontes, geração de arquivos compiláveis em Java e AspectJ.

Os modelos de entrada do *Hephaestus* são os seguintes:

- *Feature models* (FM) - descreve o domínio que é representado pelas características comuns e variáveis, bem como os relacionamentos e restrições entre essas características (*features*);

- *Instance models* (IM) - representa uma seleção de *features* que devem satisfazer as restrições do modelo (FM);
- *Product line assets* (PLA) - representa os artefatos configuráveis da LPS;
- *Configuration knowledge* (CK) - é o mapeamento das *features* com os artefatos presentes na LPS (PLA).

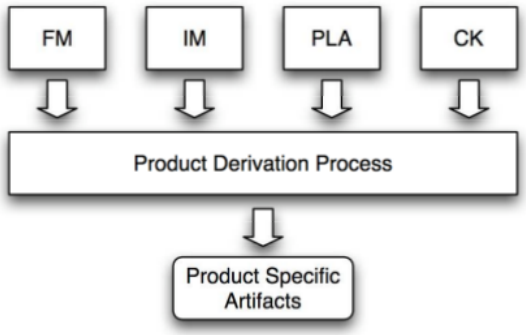


Figura 2.6: Processo de Derivação de *Hephaestus* [6]

O *Hephaestus* foi desenvolvido inicialmente para validar a abordagem de MSVCM (*Modeling Scenario Variability as Crosscutting Mechanisms*) que representa variabilidade em cenários de caso de uso [17]. Com a adição de novas funcionalidades à ferramenta, foi possível também a configuração automática de uma LPS. Com o uso de arquivos de construção (*build file*), as *features* mapeiam os arquivos fontes de entrada (classes e aspectos), por exemplo em um modelo de *features* para dispositivos móveis - *Mobile Media* com as *features*: *Photo, Sorting, Favorite*, os arquivos de construção resultantes são:

```
#Photo
src/lancs/mobilemedia/alternative/photo/PhotoAspect.aj
...
src/lancs/mobilemedia/alternative/photo/PhotoViewScreen.java
#Sorting
src/lancs/mobilemedia/option/sorting/SortingAspect.aj
#Favorite
src/lancs/mobilemedia/alternative/favourites/FavouritesAspect.java

#Sorting AND Favorite
src/lancs/mobilemedia/optional/SortingAndFavorite.aj
```

Nesse mapeamento, destaco o relacionamento entre as *features Sorting* e *Favorite* que indica o relacionamento de 1:1 entre essas *features*.

Na atual versão, a partir de cenários de caso de uso, são gerados (Figura 2.7) produtos específicos de modelos de casos de uso. Os cenários de caso de uso são descritos em

documentos de textos [17] que serão exportados para um arquivo XML, para ser *parsed* por um biblioteca do *Hephaestus*.

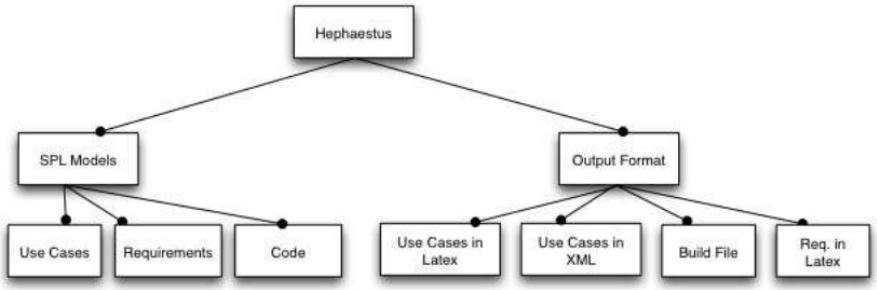


Figura 2.7: Modelo de *Features* e Arquivos de Saída do *Hephaestus* [16]

Os casos de uso especificam o comportamento de um sistema, descrevendo a funcionalidade do sistema desempenhada pelos atores. Cenário é uma sequência de passos especificando uma interação entre um usuário e o sistema. Na Figura 2.8 mostra a sequência de ações entre cliente e um sistema de compras pela Internet para realizar uma compra virtual.

id	User Action	System Response
1	Select the checkout option.	Present the items in the shopping cart and the amount to be paid. The user can remove items from the shopping cart.
2	Select the confirm option.	Request bonus and payment information.
3	Fill in the requested information and select the proceed option.	Request the shipping method and address.
4	Select one of the available shipping methods (**Economical, Fast**), fill in the destination address and proceed.	Calculate the shipping costs.
5	Confirm the purchase.	Execute the order and send a request to the Delivery System to dispatch the products.
6	-	Update the preferences based on the search results or purchased items.

Figura 2.8: Caso de Uso para Compras Virtuais [17]

O *Hephaestus* possui uma interface gráfica simples (Figura 2.9) para permitir que engenheiros possam selecionar modelos de entrada (modelo de *features*, *configuration knowledge*) e produzir produtos derivados (Figura 2.6).

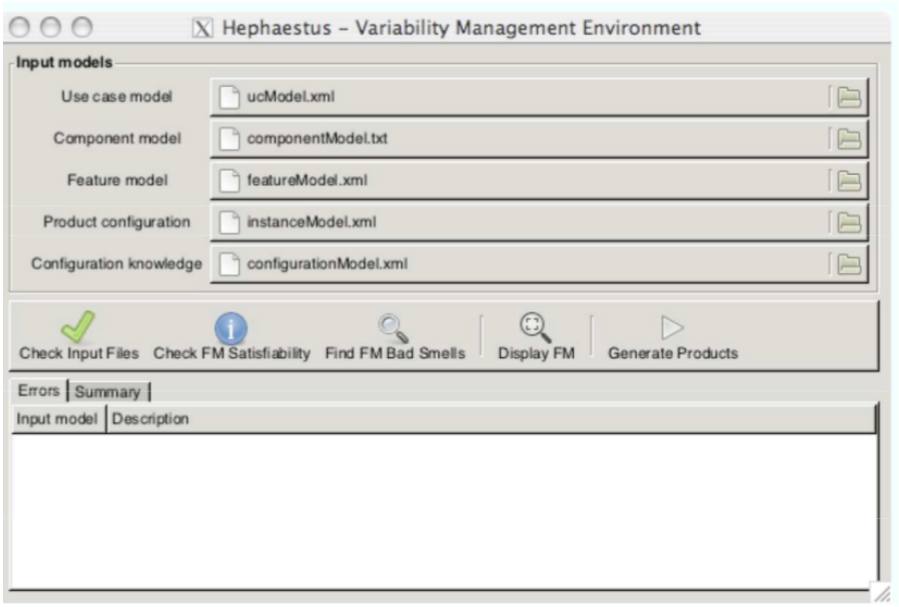


Figura 2.9: Interface do *Hephaestus* [6]

2.6.1 As Bibliotecas do *Hephaestus*

O *Hephaestus* é uma suíte de bibliotecas e ferramentas na linguagem funcional Haskell [6]. Com os seguinte módulos:

- Núcleo (*core*) - define funções e tipos de dados para todas as demais bibliotecas do *Hephaestus*.
- Modelo de *features* - define tipos de dados para a modelagem para o modelo de *features* e instâncias dos produtos derivados. Nesta biblioteca, há funções para: traduzir os modelos de *features* para lógica proposicional, checar a satisfatibilidade do modelo de *features*, identificar inconsistências (*bad smells*) no modelo de *features*.
- Modelo de caso de uso - define representação abstrata para o modelo de caso de uso para a linha de produto. Inclui definição de tipos de dados para os casos de uso, aspectos de caso de uso.
- Modelo de componente - define o mapeamento entre nomes e arquivos de código fonte. Diversos itens do *configuration knowledge* se referem a ordem de seleção dos arquivos fontes que foram incluídos no arquivo de construção (*build file*).
- *Configuration knowledge* - é responsável pelas configurações das *features* com os artefatos (*product line assets*). Mapeia uma *feature* e um artefato por meio de transformações (Figura 2.10). Implementa o mecanismo para derivação de produtos.

Feature Expression	Transformations
MobileMedia	select MainUIMidlet, ..., select MediaUtil
Photo	select PhotoAspect, ..., select PhotoViewScreen
Sorting **and** Favorite	select SortingAndFavorite
Copy **and** Video	select CopyAndVideo
...	...

Figura 2.10: Mapeamento de expressão de *features* com arquivos de construção(*build files*) [20]

O *Hephaestus* foi implementado em Haskell que é uma linguagem de programação funcional devido as facilidades inerentes à linguagem, como funções de alta ordem, funções parciais, tais aspectos da linguagem proporcionaram uma implementação simples e elegante [17].

Com as funções de alta ordem, é possível definir a família de transformação de tipo. O tipo *Transformation*, por exemplo, corresponde a qualquer função que recebe como argumento algum tipo artefato (*product line assets* - PLA). O *configuration knowlegde* é representado com um configuração de itens, sendo um par (*FeatureExpression, [Transformation]*). De modo que o processo de derivação (fragmento de código abaixo) filtra os pares válidos a fim de gerar uma instância de produto (*instance model*-IM). A função parcial possibilita a instanciação dos itens configurados, com diferentes assinaturas. O resultado das aplicações determina uma família de transformações.

```
type Transformation :: PLA -> Product -> Product
type ConfigurationKnowledge = [ConfigurationItem]
data ConfigurationItem = ConfigurationItem{
        expression :: FeatureExpression,
        transformations :: [Transformation]
}
```

O código acima define um item de configuração - `ConfigurationItem` - que possui uma expressão lógica - `expression` - representando o relacionamento entre as *features* e a lista de transformações (em Haskell uma lista é representada `[Transformation]`). Uma transformação é um tipo sinônimo que recebe artefato e um produto, e devolver um produto - `PLA -> Product -> Product`.

```
productDerivation fm im ck pla = refine tasks pla newProduct
        where
                tasks = concat [transformations c | c <- ck,
                    eval in (expression c)]
                newProduct = ...
                refine [] pla product = product
                refine (t:tx) pla product = refine ts pla (t pla product)
```

A derivação de um produto (`productDerivation`) ocorre o processamento da modelo de *features*, juntamente com a seleção de *features*, configuração e o artefato escolhido, representados respectivamente pelos argumentos de entrada: `productDerivation fm im ck pla`.

O *hephaestus* utiliza bibliotecas Haskell existentes (SYB, Parsec, HXT e gtk2HS) no repositório *Haskell* que são otimizadas e simplificaram o código fonte da ferramenta.

Capítulo 3

Análise Automática do Modelo de *Features* com *Hephaestus*

O modelo de *features* se mostra adequado para capturar as características comuns e variáveis numa linha de produtos de *software*. Todavia, há de se considerar que o número de possíveis produtos pode alcançar a quantidade 2^n (crescimento exponencial), sendo n o número de *features*, de modo que se justifica buscar uma automatização da análise de *features*.

Em um recente *survey*, Benavides *et al.* consolidam um conjunto de operações que possibilita a análise automática do modelo de *features* [7]. O objetivo deste capítulo é comparar o suporte ferramental de análise de *features* existente na ferramenta *Hephaestus* com as operações descritas em [7]. Para isso, a próxima seção apresenta o modelo de *features* representado pela ferramenta *Hephaestus*, a fim de contribuir para a melhor compreensão das implementações propostas para as análises existentes.

3.1 Representação do Modelo de *Features* no *Hephaestus*

Um modelo de *features* delimita o escopo de uma LPS, em termos de características (*features*) e o relacionamento entre as características [7]. É uma representação hierárquica (forma de árvore) das propriedades e os relacionamentos entre *features* e sub-*features* (pais e filhos). Adicionalmente, existem relacionamentos cruzados (entre sub-árvores) entre as *features*, representando restrições entre elas. A Figura 3.1 é um modelo de *features* para dispositivos móveis ilustrando as possíveis características dos produtos: um telefone celular deve obrigatoriamente possuir suporte para chamadas (*feature Calls*), possuir uma tela (*feature Screen*) opcionalmente poderá conter capacidade de mídia (*features: Media, Camera e MP3*), sendo que essas últimas *features* definem um relacionamento de grupo do tipo *ou inclusivo* o qual indica que qualquer combinação de recursos de multimídia é válida.

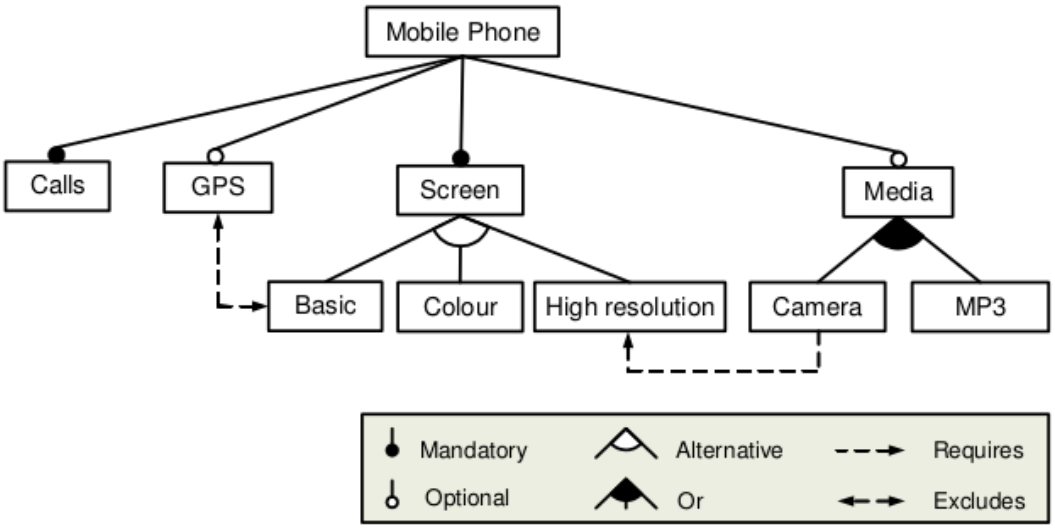


Figura 3.1: Modelo de *feature* para dispositivo móvel [7]

As *features* se relacionam entre si requerendo ou não a presença de outra *feature*, na Figura 3.1 caso a *feature* GPS seja selecionada para um produto, a *feature Basic* - uma tela básica de baixa resolução não poderá ser selecionada -, portanto sendo excluída; por outro lado, caso o dispositivo móvel possua câmera, a *feature High resolution* que representa uma tela de alta resolução deverá requerida para o novo produto.

As *features* são utilizadas para mapear as características do sistema desenvolvido em um modelo hierárquico que será utilizado no processo de análise automática. Esse processo (Figura 3.2) ocorre em duas etapas [7]: o modelo de *features* é traduzido para uma representação especifíca - lógica proposicional, programação restritiva, lógica descritiva; em seguida, utiliza-se resolvedores (*sat-solvers*) e algoritmos para selecionar alguma operação de análise automática.

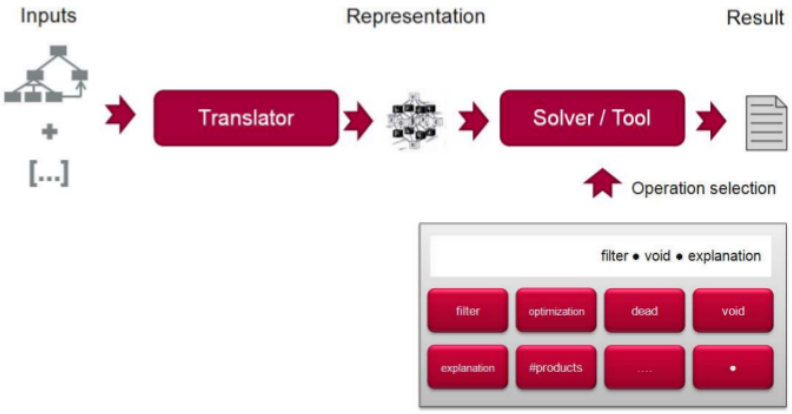


Figura 3.2: Processo para análise automática do modelo de *features* [7]

A análise do modelo de *features* é executada por meio de operações que automatizam o processo de análise. Conforme sugerido por Benavides [7], uma operação recebe um conjunto de parâmetros de entrada e retorna um resultado na saída. Dado um modelo de *features*, o conjunto das *features* (F), uma configuração escolhida é uma dupla (S,R), tal que (S,R) $\subseteq$ F, sendo S o conjunto de *features* selecionadas e R o conjunto de *features* a serem removidas. Logo S $\cap$ R = $\emptyset$. Por exemplo, o conjunto de *features* para o modelo de dispositivo móvel (Figura 3.1): F= {*Mobile Phone*, *Calls*, *GPS*, *Screen*, *Media*, *Basic*, *Colour*, *High resolution*, *Camera*,*MP3*}, se for escolhido um telefone celular com GPS e câmera, permitindo áudio com qualidade MP3, as *features* selecionadas serão - S={*Mobile Phone*, *Calls*, *GPS*, *Screen*, *Media*, *High resolution*, *Camera*, *MP3*}, sendo removidas as seguintes *features* - R={*Basic*, *Colour*}.

No *Hephaestus* as definições para o modelo de *features* encontram-se no módulo de definições do modelo *features*. Uma *feature* compreende um identificador, um nome, um tipo (indicando se a *feature* é obrigatória ou opcional), um tipo de grupo (que pode assumir os valores de *ou inclusivo*, *ou exclusivo*, ou uma indicação que *feature* não define um grupo). Essa representação em *Hephaestus* corresponde à seguinte definição:

```
data Feature = Feature {
 fId :: Id,
 fName :: Name,
 fType :: FeatureType,
 groupType :: GroupType,
 }
```

Os tipo de dados para *features* obrigatórios, opcionais, básicas e alternativas (definições fornecidas no capítulo anterior 2.4) são os seguintes:

```
data FeatureType = Optional | Mandatory

data GroupType = BasicFeature | AlternativeFeature | OrFeature
```

A notação acima define o tipo *FeatureType* sendo *Optional* ou *Mandatory*, bem como um grupo de *features* sendo dos tipos: *BasicFeature*, *AlternativeFeature* e *OrFeature*.

No *Hephaestus*, o modelo de *features* é representado por meio de uma lista expressões em lógica proposicional (`[FeatureExpression]`), por meio da função `fmToPropositionalLogic`, assim definida:

```
fmToPropositionalLogic fm = rootProposition ++ ftPropositions
    ++ csPropositions
```

O código acima indica que a função `fmToPropositionalLogic` recebe um modelo de *Features* e retorna a uma lista de expressões de *features* que é a concatenação da *feature-raiz* (`rootProposition`), com a lista de *features*-filhas (`ftPropositions`) e lista de restrições (`csPropositions`).

O mapeamento das definições do modelo de *features* - *feature*, tipo e grupo para a lógica proposicional está apresentado na Figura 3.3.

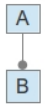
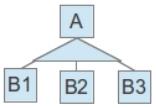
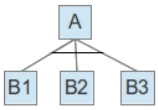
Definições do Modelo de Features	Lógica Proposicional	Representação Hephaestus
 Feature raiz	A=True	["A"]
 Relacionamento mandatório	B ↔ A	[Or(Not ("A"),"B"), Or (Not("B"), "A")]
 Relacionamento opcional	B → A	[Or(Not ("B"), "A")]
 Ou inclusivo	(A ↔ (B1 ∨ B2 ∨ B3))	[Or(Not ("A), Or("B1","B2","B3")), Or (Not (Or("B1","B2","B3")), "A")]
 Ou exclusivo	(B1 ↔ (A /\~B2 /\ ~B3)) /\ (B2 ↔ (A /\ ~B1 /\ ~B3)) /\ (B3 ↔ (A /\ ~B1 /\ ~B2))	[And(Or(Not("B1"),And("A",Not("B2"),Not("B3"))),Or(Not(And("A",Not("B2"),Not("B3")),"B1"),Or(Not("B2"),And("A",Not("B1"),Not("B3"))),Or(Not(And("A",Not("B1"),Not("B3")),"B2"),Or(Not("B3"),And("A",Not("B1"),Not("B2"))),Or(Not(And("A",Not("B1"),Not("B2")),"B3"))]
 Relacionamento de dependência	A → B	[Or(Not ("A"),"B")]
 Relacionamento de exclusão	~(A /\ B)	[Not (And ("A","B"))]

Figura 3.3: Mapeamento de definições - modelo de *features*, lógica proposicional e representação *Hephaestus*

3.2 Operações de Análise do Modelo de *Features*

Para que a linha de produtos de *software* gere produtos consistentes que obedeçam às restrições definidas na configuração escolhida é necessária a detecção de erros. As operações descritas a seguir buscam detectar anomalias já no modelo de *features*. Essas operações recebem como entrada o modelo de *features* retornando as inconsistências observadas.

Op.1 *Feature* vazia - (*Void feature*) - é uma operação para identificar inconsistências devido a restrições contraditórias entre as *features*. Na Figura 3.4 apresenta a inconsistência entre as *features* filhas B e C visto que são obrigatórias, mas que não poderão aparecer em nenhum produto.

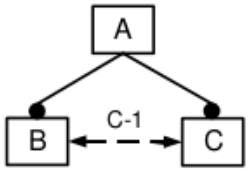


Figura 3.4: Modelo de *feature* vazia [7]

No *Hephaestus*, a principal função que verifica inconsistências no modelo de *features* é a operação *fmTypeChecker*, assim definida:

```
fmTypeChecker :: FeatureModel -> CheckResult
fmTypeChecker fm =
 case errors of
  [] -> Success
  xs -> Fail {errorList = erros}
```

A operação recebe um modelo de *features* de entrada, indicando *Success* se não há inconsistência no modelo, ou *Fail {errorList = errors}* - falha e lista de erros.

Op.2 Produto válido - é uma operação para verificar se um produto pertence ao conjunto de produtos gerados a partir de um modelo *features*. Em relação ao modelo para dispositivo móvel, o produto P={*MobilePhone,Screen, Colour, Media,MP3*} não pertenceria ao conjunto de produtos gerados pelo modelo *Mobile Media* um vez que não consta a *feature* obrigatória *Calls*. É uma checagem de configuração.

No *Hephaestus*, a operação que avalia se um produto é válido é a função *validInstance*, assim representada:

```
validInstance :: FeatureModel -> FeatureConfiguration -> Bool
validInstance fm fc =
 let fmExpression = foldAnd(fmToProsicionalLogic fm)
 in eval fc fmExpression
```

Essa operação recebe um modelo de *features* e uma configuração de *features* selecionada, retornando um valor booleano que será verdadeiro se a configuração escolhida estiver consistente com o modelo de entrada. Observa-se que o modelo de *features* é convertido inicialmente para lógica proposicional (por meio da função *fmToPropositionalLogic*), para em seguida ser avaliado com a configuração selecionada (`eval fc fmExpression`).

Op.3 Configuração parcial válida - esta operação recebe um modelo de *feature* e uma configuração parcial e informa se a configuração é válida.

Op.4 Todos os Produtos - é uma operação que visa identificar todos os possíveis produtos de um modelo de *features* e consequentemente calcular o número desses produtos.

Op.5 Número de produtos - esta operação informa o número de produtos que podem ser gerados a partir do modelo de *features* considerando a flexibilidade e complexidade da LPS.

Op.6 Filtro - recebe um modelo de *feature* e uma configuração retornando um conjunto de produtos que podem ser derivados daquele modelo. É importante destacar que tal operação não modifica o modelo, apenas filtra produtos a partir da configuração escolhida.

Op.7 *Dead features* - esta operação identifica as *features* que não aparecerão em nenhum produto derivado. Ocorre devido a erro de relacionamento entre *features*. Na Figura 3.5 a *feature* em cinza não aparecerá em nenhum produto tendo em vista a restrição presente no modelo.

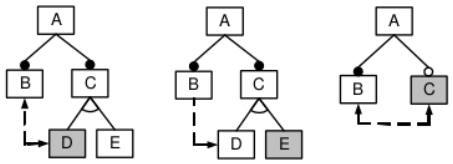


Figura 3.5: Exemplo de modelos com *dead feature* [7]

Op.8 *Features* condicionalmente mortas(*dead*) - uma *feature* pode ficar condicionalmente morta sob certas condições. Na Figura 3.6, a *feature* B se tornará morta sempre que a *feature* D for selecionado.

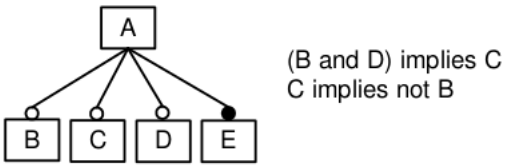


Figura 3.6: Um exemplo de *feature* condicionalmente morta [7]

Op.9 *Feature* opcional falsa - trata-se de uma *feature* presente em todos os modelos, devendo ser classificada como obrigatória, entretanto é modelada como *feature* opcional. Na Figura 3.7 as *features* requeridas pela *feature* obrigatória estarão sempre presentes nos produtos, logo elas deveriam ser classificadas como *features* obrigatórias.

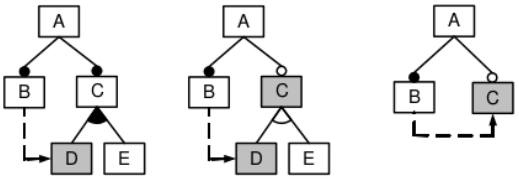


Figura 3.7: Falsas *features* opcionais [7]

Op.10 Erro de cardinalidade - ocorre quando uma cardinalidade prevista não puder ser constatada devido a presença de certas restrições. Na Figura 3.8 está prevista uma cardinalidade entre 1 e 3 *features*, entretanto devido ao relacionamento de exclusão entre as *features* B e D, à cardinalidade correta deveria ser entre 1 e 2.

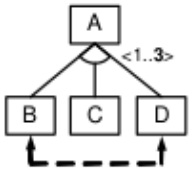


Figura 3.8: Erro de cardinalidade num modelo de *features* [7]

Op.11 Redundâncias - ocorre quando a mesma informação semântica é modelada de forma diversas. Na Figura 3.9, por exemplo, no primeiro modelo, o grupo já é do tipo exclusivo, não há necessidade de definir um relacionamento transversal (entre as subárvores) também exclusivo.

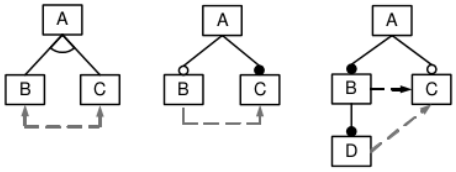


Figura 3.9: Exemplos de redundâncias num modelo de *features* [7]

Op.12 Explanações - esta operação recebe o modelo de *features*, informando a seguir (saída) as causas que provocam as anomalias, por exemplo uma possível justificativa para que a *feature* D está morta devido à restrição de exclusão com a *feature* B que é obrigatória (sempre aparece em qualquer produto gerado, Figura 3.10). Ao detectar as inconsistências no modelo *features*, os engenheiros de domínio devem proceder a necessária correção seja pelo remoção de relacionamentos não permitidos (se possível) ou realizando uma reestruturação de todo o modelo.

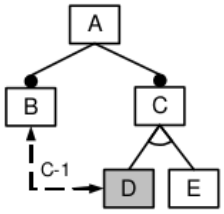


Figura 3.10: Explicação para *feature* morta [7]

Op.13 Explanações corretivas - esta operação recebe um modelo de *feature*, tendo como resultado um conjunto de explicações indicando as mudanças (possíveis sugestões) necessárias para corrigir a inconsistência detectada. Na Figura 3.10, poderia ser proposto a retirada da exclusão entre as *features* B e D, ou transformando a *feature* B em opcional.

Op.14 *Features* centrais - é uma operação que informa o conjunto de *features* presentes em todos os produtos.

Op.15 *Features* variáveis - identifica as *features* variáveis para um dado modelo de *features*.

Op.16 Conjuntos atômicos - esta operação recebe o modelo de *features* e informa os conjuntos atômicos. Um conjunto atômico é um grupo de *features* que deve ser tratado com uma unidade. A Figura 3.11 ilustra essa operacao.

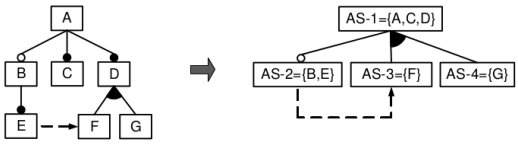


Figura 3.11: Cálculo de um conjunto atômico [7]

Op.17 Análise de dependência - esta operação recebe como entrada o modelo de *features* e um configuração parcial, retornando uma nova configuração com *features* que devem ser selecionadas ou removidas. É uma operação a fim de propagar alterações e restrições durante uma definição de modelo de *features* interativa.

Op.18 Configuração com vários passos - é a produção de uma série de configurações a partir de um modelo de *features* e uma configuração parcial. Esta operação examina uma lista de possíveis passos a partir de uma configuração inicial com intuito de gerar um modelo bem formado.

Op.19 Grau de homogeneidade - esta operação informa o grau de homogeneidade do modelo de *features*, isto é, se um modelo tem poucas *features* únicas então ele é mais homogêneo. *Features* únicas (*unique features*) são as que aparecem somente em um único produto.

A fórmula é: homogeneidade $= 1 - \frac{\#uf}{\#products}$, sendo #uf o número de *features* únicas e $\#products$ denota o número de produtos possíveis com o modelo de *features* de teste.

Op.20 Produtos comuns - esta operação recebe um modelo de *features* e um configuração, informando em seguida o percentual de produtos derivados desses parâmetros de entrada.

Op.21 Fator de variabilidade - esta operação recebe como entrada o modelo de *features* e retorna a taxa entre o número de produtos e o número de *features*. É um indicador para medir a flexibilidade do modelo.A fórmula é: $\frac{NumerodeProdutos}{2^n}$.

Op.22 Grau de ortogonalidade - informa a taxa entre o número de produtos de um modelo de *features* e o número de produtos de uma subárvore. Um alto grau de ortogonalidade indica a independência entre as partes da árvore.

Op.23 Restrições extra-representativas - esta operação indica o grau de representatividade entre as restrições de uma árvore. É a relação entre o número de *features* interrelacionadas e o número de *features* da árvore.

Op.24 Ancestral comum - recebe como entrada o modelo de *features* e um conjunto selecionado, retornando uma *feature* que é o ancestral comum daquele conjunto.

Op.25 *Feature* raiz - esta operação informa o conjunto de raízes(*roots*) para um modelo de *features* e um conjunto de *features* selecionado.

O quadro (Tabela 3.1) a seguir apresenta o comparativo entre as operações de análise automática de *features* propostas por Benavides e as operações que estão implementadas na ferramenta *Hephaestus*.

Tabela 3.1: Comparativo entre as Operações para Análise Automática e *Hephaestus*

Operações	**Suporte no *Hephaestus***
Op.1 - *Void Feature*	sim
Op.2 - Produto Válido	sim
Op.3 - Configuração parcial válida	não
Op.4 - Todos os Produtos	não
Op.5 - Número de Produtos	não
Op.6 - Filtro	sim
Op.7 - *Dead Features*	sim
Op.8 - *Features* Condicionamento Mortas	não
Op.9 - *Feature* False	não
Op.10 - Erro de Cardinalidade	não
Op.11 - Redundâncias	parcialmente
Op.12 - Explanações	parcialmente
Op.13 - Explanações corretivas	não
Op.14 - *Features* centrais	sim
Op.15 - *Features* Variáveis	não
Op.16 - Conjuntos Atômicos	não
Op.17 - Análise de dependência	não
Op.18 - Configuração com vários passos	não
Op.19 - Grau de homogeneidade	não
Op.20 - Aspectos comuns	não
Op.21 - Fator de variabilidade	não
Op.22 - Grau de ortogonalidade	não
Op.23 - Restrições representativas extras	não
Op.24 - Ancestral comum	não
Op.25 - *Feature* raiz	não

Capítulo 4

Novas Operações do Módulo de *Features*

A partir das Tabela 3.1 foram selecionadas as seguintes operações objetivando melhorar a escalabilidade da análise automática do modelo de *features* da ferramenta *Hephaestus*:

- Número de produtos.
- Grau de homogeneidade.
- Fator de variabilidade.
- Produtos comuns.
- Grau de ortogonalidade.

Além dessas operações acima propostas no *survey* de Benavides et al. [7], também foram implementadas funções para identificar os erros na estrutura do modelo de *features* fornecido e na configuração escolhida de um modelo.

As novas operações foram agrupadas considerando dois aspectos de utilidade: mensuração de propriedades - número de produtos, homogeneidade e produtos comuns; e, verificação de inconsistências no modelo e na configuração - teste de satisfatibilidade com diagramas binários de decisão (OBDD), erros no modelo, explanação e *features* não pertencentes ao modelo.

4.1 Operações Mensuráveis

Neste primeiro grupo de operações constam funções para realizar o cálculo do número de produtos possíveis para o modelo de *features*, grau homogeneidade, fator de variabilidade e número de de produtos comuns. Para a construção das funções, foi utilizada a biblioteca Haskell que implementa diagramas binários de decisão (OBDD). As funções da biblioteca foram adaptadas para uso no modelo de *features* realizando a conversão do modelo de *features* da ferramenta *Hephaestus* da seguinte forma:

```
featureExpressionToOBDD :: FeatureExpression -> O.OBDD FeatureExpression
featureExpressionToOBDD f = case f of
 (And e1 e2) ->
```

```
        O.and [(featureExpressionToOBDD e1), (featureExpressionToOBDD e2)]
 (Or e1 e2) ->
        O.or [(fatureExpressionToOBDD e1), (featureExpressionToOBDD e2)]
 (Not e1) ->
        O.not (featureExpressionToOBDD e1)
  (FeatureRef id1) ->
        O. unit (FeatureRef id1) True
```

A função `featureExpressionToOBDD` recebe uma expressão de *feature* como argumento e de acordo com a expressão (`And, Or, Not`) ela é convertida recursivamente para o formato OBDD. Devendo a expressão de *feature* estar no formato disjuntivo normal.

Número de produtos - essa operação determina o número de produtos possíveis a partir de um modelo de *features*, levando em conta tanto o número de *features* existentes quanto das restrições (*constraints*) presentes no modelo em análise.

```
numberOfModels :: FeatureModel -> Integer
numberOfModels fModel =
        O.number_of_models (Set.fromList (fList fModel)) (fOBDD fModel)
```

Além da conversão `fOBDD::FeatureExpression -> O.OBDD FeatureExpression` anteriormente mencionada, também foram extraídas as expressões de *features* presentes no modelo por meio da função `fList :: FeatureModel -> [FeatureExpression]` . As expressões booleanas ficaram no formato conjuntivo normal (CNF).

```
fList :: FeatureModel -> [FeatureExpression]
fList fModel = getVars (fmToCNFExpression fModel)
```

Grau de homogeneidade - a operação identifica a quantidade de *features* únicas (*unique features*) que aparecem apenas em um único produto. Um modelo de *features* será mais homogêneo caso tenha poucas *features* únicas. Quando ocorre de um modelo possuir muitas *features* únicas, esse modelo será heterogêneo. O grau de homogeneidade é dado pela fórmula
homogeneidade = 1 - $\frac{\#uf}{\#products}$. O cálculo do grau de homogeneidade é feito na função `homogeneity` a qual retorna um valor em ponto flutuante.

```
homogeneity :: FeatureModel -> Float
homogeneity fm = 1.0 - (fromIntegral ufs / fromIntegral nps)
 where
  ufs = length $ uniqueFeatures fm
  nps = numberOfModels fm
```

A variável `ufs` receberá o comprimento da lista de *features* únicas por meio da função `uniqueFeatures fm`. A variável `nps` receberá a quantidade de produtos gerados pela função `numberOfModels`.

Produtos Comuns - reutilizando a função para identificar produtos com *features* únicas - `uniqueFeatures` foi possível criar a operação *commonality* (produtos comuns) a fim de verificar o número de produtos com uma mesma configuração de *features*. Observa-se que essa operação, além de receber com entrada o modelo de *features*, também receberá uma configuração parcial. O código resultante foi:

```
commonality :: FeatureModel -> FeatureConfiguration -> Float
commonality fm@(FeatureModel _ cs) fc =
  fromIntegral totalConfig / fromIntegral totalModels
```

Fator de Variabilidade - essa operação apresenta a razão do número de produtos possíveis originados a partir de um modelo de *features* e a quantidade de 2^n, sendo 'n' número de *features* existentes. A implementação da operação segue no fragmento de código abaixo.

```
variabilityFactor :: FeatureModel -> Float
variabilityFactor fm = fromIntegral totalModels /nFeatures
 where
  totalModels = numberOfModels fm
  nFeatures = 2 ^ (length (fmList fm))
```

4.2 Verificação de Inconsistências de Modelo e Configuração de *Features*

Nesta seção, foram agrupadas as operações que visam analisar a estrutura do próprio modelo de *features* e a configuração escolhida. Essas operações identificam os relacionamentos presentes que provocam inconsistências no modelo. A ferramenta *Hephaestus* já permitia a análise de configuração escolhida, por meio da da operação *validInstance*, informando o usuário se a configuração é válida, entretanto não apresentava na saída os relacionamentos entre as *features* que causavam a inconsistência.

Void features - verifica se o modelo é insatisfatível. Essa verificação já era feita por meio da função - *fmTypeChecker*. Com a bibilioteca OBDD pretendeu-se acrescentar ao módulo de *features* uma implementação com diagramas binários de decisão, objetivando a diminuição do tempo de resposta no teste de satisfatibilidade do modelo de *features* fornecido. A função `fmSATOBDD` recebe o modelo de *features* e retorna um tipo booleano que será verdadeiro, se o modelo seja satisfatível. O código é o seguinte:

```
fmSATOBDD :: FeatureModel -> Bool
fmSATOBDD fm = satisfiable fmToOBDD
```

Análise do modelo - essa operação recebe um modelo de *features* de entrada e, caso o modelo seja insatisfatível, retorna a lista de relacionamentos entre as *features* que provoca tal insatisfatibilidade. O código é o seguinte:

```
modelAnalysis :: FeatureModel -> [FeatureExpression]
modelAnalysis fm =
    [ x | x<-cs, fmTypeChecker (FeatureModel ft [x]) /= Success]
```

A função utiliza uma lista de compreensão *Haskell* que analisa a lista de restrições (cs - *constraints*) de acordo com o predicado `fmTypeChecker (FeatureModel ft [x]) /=Success`, sendo ft, a representação da árvore de *features*. Por exemplo, seja submetendo o modelo da Figura 4.1 (código representado abaixo), o resultado será a lista `[Not (And ("b", "e"))]` que representa a lista de relacionamentos que provocou a insatisfatibilidade do modelo de *features* fornecido para a função `modelAnalysis`.

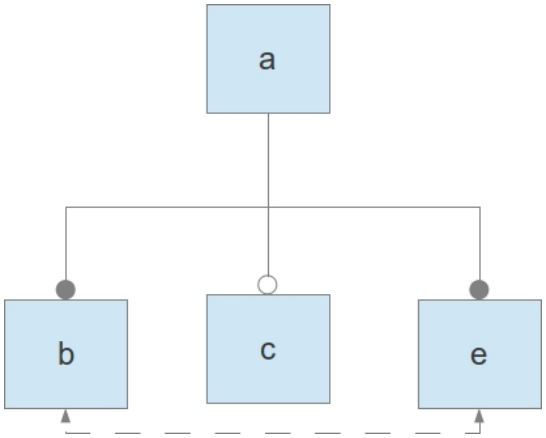


Figura 4.1: Utilização da análise de modelo

```
modelo17=FeatureModel{
 fmTree=Root a [Leaf b, Leaf c, Leaf e],
 fmConstraints=[Not (And (FeatureRef "b") (FeatureRef "e"))]
}
```

Explanações - nesta operação procurou-se identificar as possíveis falhas de uma configuração de entrada em comparação com o modelo de *features* dados, indicando por meio de expressões proposicionais as incompatibilidades entre a configuração escolhida e o modelo de *features* dados. Por exemplo, sendo apresentado o modelo abaixo - modelo10 (Figura 4.2): as *features*: a e b são *features* obrigatórias, c e i são *features* opcionais, sendo que a *feature* i não poderá ser escolhida junto com a *feature* c (são *features* exclusivas).

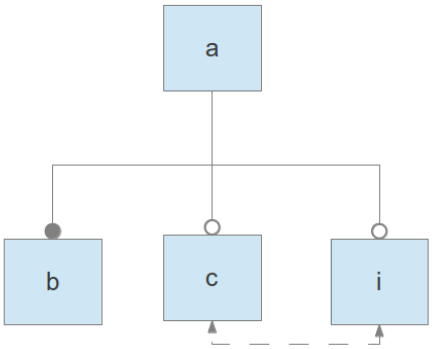


Figura 4.2: Exemplo de Modelos de *Features* - Modelo10

```
modelo10 = FeatureModel{
    fmTree=Root a [Leaf b, Leaf c, Leaf i],
    fmConstraints=[Not (And (("i"),("c")))]
}
```

Se for escolhida a configuração `FeatureConfiguration {ft=Root a [Leaf b, Leaf c,Leaf i]}` (
gura 4.3), a operação *explanation* indicará a lista de restrições que não está sendo respeitada, na configuração exemplificada será: `[Not(And(("i"), "c"))]` .

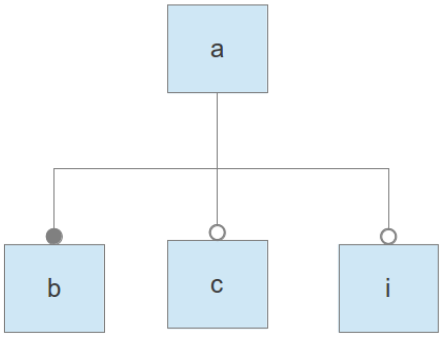


Figura 4.3: Configuração de *features* baseada no modelo 10 (Figura 4.2)

Verificação de *Features* Inexistentes - essa operação verifica se as *features* constantes na configuração estão de fato presentes no modelo de *features* de entrada. No modelo acima mencionado, caso seja escolhida uma configuração:

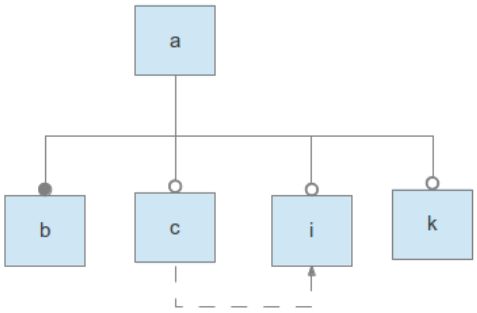


Figura 4.4: Configuração de *features* baseada no modelo 10 (Figura 4.2)

`FeatureConfiguration{ft=Root a [Leaf b, Leaf c, Leaf i, Leaf k}`

Observa-se que a *feature* k (Figura 4.4) não pertence ao modelo fornecido, a operação de *checkFeatures* apresentará a lista de *features* que não pertence ao modelo, nesse caso `[k]`.

Capítulo 5

Análise dos Resultados

Neste capítulo são apresentados os resultados dos testes realizados sobre as novas operações implementadas para o módulo de *features* da ferramenta *Hephaestus*. Os modelos utilizados nos testes são provenientes do projeto FAMA [1]. São exemplos de modelos de *features* gerados para a ferramenta - *FaMa Tool Suite (FaMaTS)*. Tais modelos foram convertidos para um formato aceito para o *Hephaestus*.

No total, 52 modelos de *features* foram analisados. Esses modelos apresentam características diferentes variando em número de *features* de 10 a 137 *features*, com número de restrições (*constraints*) entre 14 a 179 restrições. Foram computados o tempo de execução e resultados das novas funções do módulo de *features*. Os dados foram obtidos de 10 repetições para cada modelo. O *hardware* utilizado foi um computador com a seguinte configuração: processador Intel DC E5700, memória RAM com 4 GB, sistema operacional Ubuntu 12.04, *kernel* 3.2.0.45.

Foram avaliadas as seguintes operações: verificação de satisfatibilidade (SAT) com diagramas binários de decisão (OBDD), geração de número de produtos de *features*, verificação do grau de homogeneidade e verificação do fator de variabilidade no modelo de *features*.

Em linhas gerais, os testes descritos neste capítulo objetivam verificar a escalabilidade da análise do modelo de *features* considerando a implementação das novas operações. Os testes foram realizados executando cada modelo de *features* de acordo com a operação a ser testada (satisfatibilidade, número de produtos, etc) armazenando os dados de tempo para extrair as informações relevantes.

5.1 Verificação de satisfatibilidade com diagramas binários de decisão

A análise da satisfatibilidade do modelo de *features* já era realizada com a biblioteca Funsat. A nova operação com diagramas binários de decisão foi proposta para reduzir o tempo de execução para essa análise. Neste cenário, foram avaliadas as seguintes hipóteses:

- Hipótese nula - H_0 - $\mu_{OBDD} = \mu_{Funsat}$: os tempos médios de execução entre as implementações com as duas técnicas (Funsat/OBDD) são iguais.

[1] http://code.google.com/p/famats

- Hipótese alternativa - H_1 - $\mu_{OBDD} \neq \mu_{Funsat}$: os tempos médios de execução entre as implementações com as duas técnicas são diferentes.

A tabela 5.1 apresenta as primeiras amostras de tempo medidas com dez repetições das implementações com as bibliotecas Funsat e OBDD (diagramas binários de decisão) em decorrência da análise de satisfatibilidade para modelos de *features* entre 10 e 137 *features* (a tabelas completa consta no Anexo A.1).

Tabela 5.1: Tempo de Execução das implementações de Sat-Solver Funsat (F) e OBDD (O) (tempo em segundos)

Mod.	Num	Téc.	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
2	10	F	0,008	0,009	0,008	0,01	0,009	0,009	0,009	0,009	0,009	0,009
2	10	O	0,013	0,015	0,014	0,016	0,017	0,019	0,014	0,016	0,013	0,014
8	10	F	0,009	0,01	0,009	0,01	0,009	0,01	0,01	0,01	0,01	0,009
8	10	O	0,016	0,016	0,017	0,016	0,016	0,017	0,017	0,016	0,016	0,016
12	10	F	0,006	0,007	0,006	0,007	0,007	0,006	0,007	0,008	0,006	0,007
12	10	O	0,018	0,01	0,011	0,01	0,012	0,01	0,011	0,01	0,01	0,011
17	10	F	0,006	0,006	0,007	0,007	0,008	0,007	0,007	0,008	0,006	0,007
17	10	O	0,008	0,008	0,009	0,008	0,018	0,012	0,011	0,012	0,01	0,011
21	10	F	0,005	0,004	0,005	0,004	0,005	0,006	0,004	0,005	0,005	0,005
21	10	O	0,009	0,01	0,009	0,01	0,011	0,011	0,011	0,011	0,012	0,011
30	10	F	0,004	0,005	0,005	0,004	0,004	0,005	0,005	0,004	0,004	0,004
30	10	O	0,007	0,007	0,007	0,008	0,008	0,008	0,007	0,008	0,008	0,007

A partir dos dados obtidos contidos na Tabela A.1, foi realizado o teste análise de variância (ANOVA) a fim de verificar a influência do número de *features* e técnica - OBDD e Funsat, sobre o tempo de execução. Esses resultados também foram avaliados com os seguintes testes estatísticos - teste do sinal e teste t-pareado. O teste ANOVA é utilizado para decidir se duas médias amostrais podem ser atribuídas ao acaso, ou se são indicativas de diferenças reais entre as médias das populações amostradas [21]. O teste de sinal verifica a probabilidade de obter um valor amostral diferente da mediana [21]. O teste t-pareado faz comparação de duas médias.

O teste ANOVA para a análise do tempo de execução das implementações Funsat e OBDD resultou na tabela 5.2. Nos resultados se verifica a forte influência do número de *features* e da técnica na diferença entre as média que resulta na rejeição da hipótese nula.

Tabela 5.2: Influência do número de *features* e da técnica sobre o tempo de execução(tempo em segundos)

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Features	25	8.459	0.3384	3.693	5.56e-06 ***
Tecnica	1	0.534	0.5335	5.824	0.0182 *
Residuals	77	7.054	0.0916		

O teste de sinal compara as médias da tabela 5.3 resultando um $p-value = 2.354.10^{-14}$, bem inferior do nível de significância - 5%. Por fim, foi realizado o teste t-pareado cujo resultado - $p - value = 0,0594$ -, indica um valor próximo ao nível de significância, reforçando a rejeição H_0.

Tabela 5.3: Amostra da tabela (A.2) do tempo de execução Funsat e OBDD

Modelo	*Features*	Restrições	$\mu_{Funsat(s)}$	$\mu_{OBDD(s)}$
0	10	15	0,0048	0.01050
10	10	17	0,0089	0.01510
15	10	16	0,0096	0.01630
21	10	15	0,0066	0.01220
22	10	15	0,0069	0.01070
23	10	14	0,0092	0.01330
27	10	15	0,0098	0.01650
29	10	16	0,0044	0.00750
30	10	19	0,0087	0.01390
43	10	16	0,0062	0.01200
48	10	15	0,0067	0.01130
50	10	16	0,005	0.00540
28	11	19	0,0064	0.01220
37	11	16	0,0086	0.01250
42	11	20	0,0089	0.01520
2	12	16	0,0092	0.01340

5.2 Geração do número de produtos

O número de produtos é obtido a partir do modelo de *features*, com a utilização da função *numberOfModels* que pertence à biblioteca OBDD. Esse número informa a quantidade de produtos possíveis para um modelo de *features* fornecido na entrada da função. A tabela 5.4 apresenta o resultado do tempo de execução e o número de produtos na computação dos 52 modelos de teste. Esses resultados são ilustrados também nos gráficos 5.1 e 5.2.

Na tabela 5.4 pode se verificar que o número de produtos não tem dependência linear com o número de *features* e restrições, visto que para mesmos valores de *features* e restrições, o número de produtos resultante é diferente, por exemplo, nos modelos indicados pelo índices 17 e 21, possuem o mesmo número de *features* e de restrições (10 e 15), entretanto o número de produtos para esses modelos são respectivamente 31 e 17.

Tabela 5.4: Amostra da Tabela A.3 com número de produtos de *features*

Modelo	*Features*	Restrições	$\mu_{Tempo(s)}$	Número de Produtos
2	10	17	0,0145	9
8	10	16	0,0169	16
12	10	15	0,0112	24
17	10	15	0,0116	31
21	10	15	0,017	17
30	10	16	0,0095	16
32	10	15	0,0122	16
36	10	19	0,0138	1
40	10	15	0,0165	20
43	10	16	0,0169	16
45	10	16	0,0055	12

Mesmo com a correlação de 65% entre o número de *features* e restrições em relação ao número de produtos, o gráfico 5.1 ilustra bem que essas quantidades não são linearmente dependente.

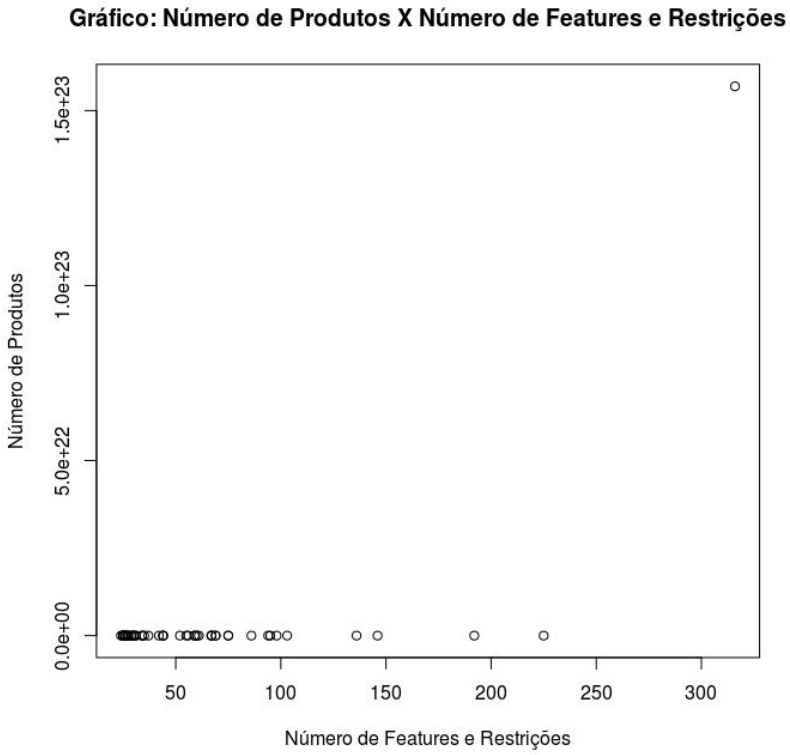


Figura 5.1: Relação entre o número de produtos e o número de *features* e restrições dos modelos de teste

Por outro lado, o gráfico 5.2 já exibe alguma tendência de dependência linear entre o número de *features* e restrições e o tempo de execução obtido com a computação dos modelos de teste. A correlação obtida entre essas quantidades foi de 67% foi até menor que o gráfico anterior.

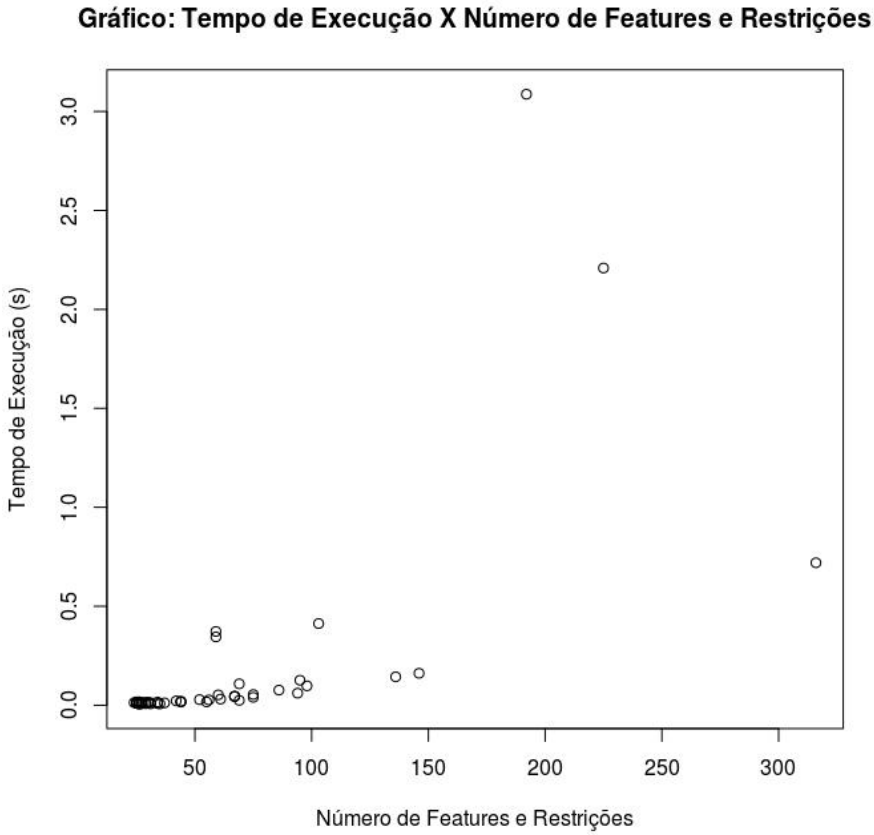


Figura 5.2: Relação entre o tempo de execução do número de produtos e o número de features e restrições

5.3 Verificação do grau de homogeneidade

O grau de homogeneidade indica a quantidade de *features* únicas presentes em um produto (definiçao 19). A tabela 5.5 apresenta a computação para os 52 modelos de teste. Em todos os modelos de teste, o valor encontrado 1, aponta que os produtos gerados não possuem tal característica.

Tabela 5.5: Amostra da Tabela A.4 dos Produtos de *Features*

Modelos	*Features*	Restrições	$\mu_{Tempo(s)}$	Homogeneidade
2	10	17	0,0507	1
8	10	16	0,0889	1
12	10	15	0,0347	1
17	10	15	0,0362	1
21	10	15	0,0995	1
30	10	16	0,0466	1
32	10	15	0,0356	1
40	10	15	0,0496	1
43	10	16	0,0653	1
45	10	16	0,0185	1

O gráfico 5.3 exibe a computação do cálculo do grau de homogeneidade dos modelos utilizados no teste. A correlação obtida entre o número de *features* e o tempo de execução do cálculo foi de 75%.

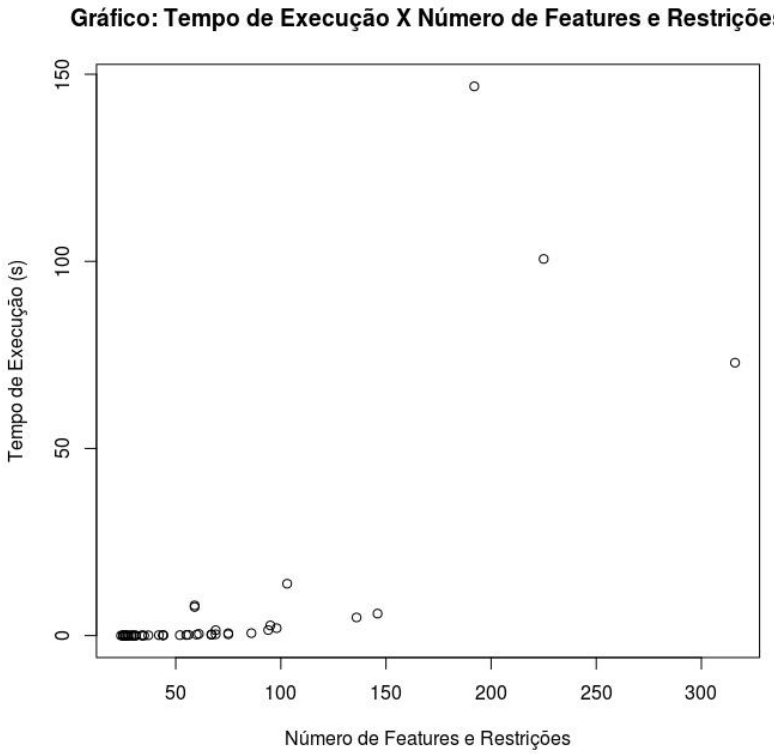


Figura 5.3: Relação entre o tempo para execução e o número de *features* e restrições no cômputo do grau de homogeneidade dos produtos

5.4 Verificação do fator de variabilidade

É a relação entre o número de produtos possíveis obtido com a função geradora do número de produtos de *features* e 2^n, sendo 'n' o número de *features* existentes no modelo de entrada.

Tabela 5.6: Amostra da Tabela A.5 dos Modelos de *Features*

Modelo	*Features*	Restrições	$\mu_{Tempo(s)}$	Fator de Variabilidade
2	10	17	0,0148	0,00879
8	10	16	0,0079	0,01563
12	10	15	0,0118	0,02344
17	10	15	0,0115	0,03027
21	10	15	0,0099	0,0166
30	10	16	0,0126	0,01563
32	10	15	0,0123	0,01563
36	10	19	0,0139	0,00098
40	10	15	0,0077	0,01953
43	10	16	0,0172	0,01563

O gráfico 5.4 apresenta a relação entre o tempo de execução e o número de *features* e restrições para o cálculo do fator de variabilidade dos modelos. A correlação obtida entre as variáveis foi de 67%.

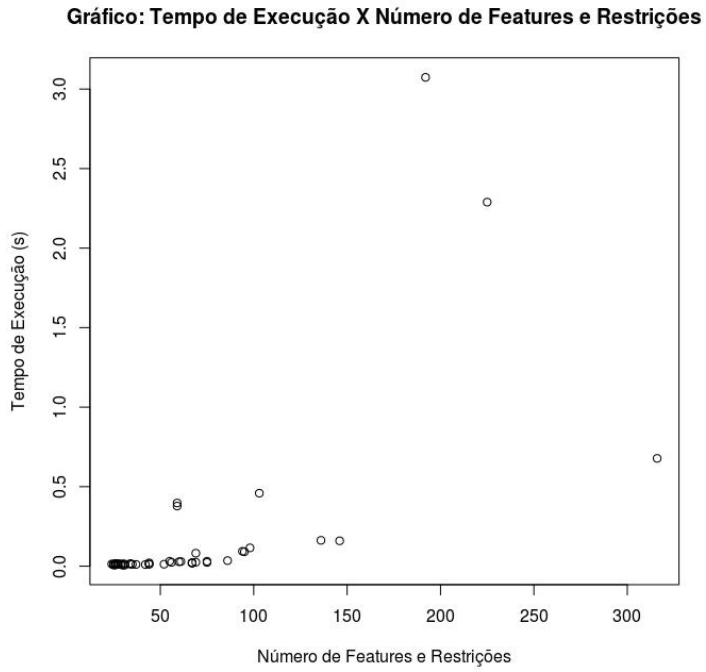


Figura 5.4: Relação entre o tempo de execução e o número de *features* e restrições para o cálculo de fator de variabilidade dos modelos de teste

5.5 Verificação da análise de modelo

A operação análise de modelo verifica a consistência do modelo de *features* objetivando explicitar caso ocorra, os erros na construção do modelo. A tabela 5.7 apresenta a amostra dos primeiros resultados para o teste realizado com 92 modelos.

Tabela 5.7: Amostra da Tabela A.6 da análise de modelos de *features*

Modelo	*Features*	Restrições	$\mu_{TempodeExecuo(s)}$
0	10	15	0,0124
14	10	17	0,004
22	10	16	0,0104
32	10	15	0,0051
37	10	15	0,0055
38	10	14	0,0041
40	10	19	0,0058
43	10	16	0,0068

A correlação obtida entre o tempo de execução e número de *features* foi de 41%, o gráfico 5.5 exibe os resultados encontrados.

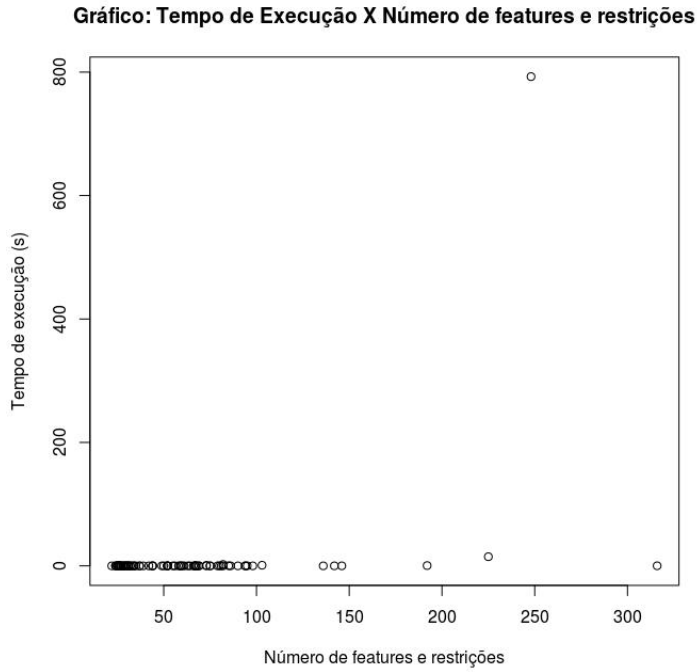


Figura 5.5: Relação entre o tempo de execução e o número de *features* e restrições na operação de análise de modelo de *features*

5.6 Conclusões

Observa-se nos resultados que as características de número de produtos gerados, grau de homogeneidade e fator de variabilidade não possuem dependência linear com o número de *features*/restrições de entrada fornecido nas funções.

Quanto à escalabilidade dessas operações na ferramenta *Hephaestus*, a fim de verificar o aumento de carga suportada para análise de satisfabilidade, foram testados, adicionalmente, modelos com 100, 200 e 500 *features*, e constatou-se que a implementação com a biblioteca Funsat suportou o modelo de até 500 *features*, enquanto que à implementação com OBDD respondeu até 100 *features*.

Capítulo 6

Conclusão e Desafios Futuros

A linha de produtos de *software* (LPS) surge como mais uma opção para aumentar a produtividade do desenvolvimento de sistemas por meio da utilização sistemática do reuso. É um grande desafio na área de pesquisa em Engenharia de *Software*. A modelagem dos sistemas por meio de *features* apoia a LPS. As ferramentas desenvolvidas para LPS possibilitam o mapeamento das funcionalidades em *features* com a utilização do modelo de *features*.

Ao longo deste trabalho foram consultados artigos que investigam a utilização do modelo de *features* para desenvolvimento da LPS, na principal referência deste trabalho - *Automated Analysis of Feature Models 20 Years Later* -, de Benavides [7] foram identificadas novas operações, principalmente de características mensuráveis (geração de número de produtos, homogeneidade, etc), que foram adicionadas à ferramenta *Hephaestus*. Com essas novas operações, foi possível computar resultados para modelos de *features* de entrada.

Os resultados dos testes mostraram que, para a análise de satisfatibilidade dos modelos de *features*, a biblioteca Funsat se apresentou com um desempenho superior à biblioteca OBDD (diagramas binários de decisão), entretanto com a utilização desta biblioteca foi possível contabilizar as características mensuráveis para o modelo de *features*.

Outra melhoria adicionada à ferramenta *Hephaestus* foram as operações para identificar falhas na configuração : Análise do modelo, Explanações e Verificações de *Features* Inexistentes, também objetivaram dotar à ferramenta com funções corretivas explícitas para a análise do modelo de *features* que permite identificar erros no modelo de *features* e no produto escolhido.

A partir deste trabalho, é possível perceber mais possibilidades de estudo e desenvolvimento neste tópico de análise automática do modelo de *features*: melhorar a escalabilidade para a análise de satisfatibilidade do modelo de *features* e tornar mais amigável para o operador a identificação de inconsistências no modelo de *features*.

O artigo de Benavides [7] aponta também a necessidade de formalizar as operações de análise a fim de facilitar a comunicação entre a comunidade científica, sendo como desafios de pesquisa: descrição das operações de análise e formalização de um *framework* para novas operações (análise de cardinalidade); inclusão dos atributos de *features* na operação de análise; análise de complexidade das operações descritas; e, desenvolvimento de *benchmarks*.

Apêndice A

Tabelas

Tabela A.1: Tempo de execução para modelos de 10 a 137 *features* (tempo em segundos, F-Funsat, O-OBDD)

Mod.	Num	Téc.	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
2	10	F	0,008	0,009	0,008	0,01	0,009	0,009	0,009	0,009	0,009	0,009
2	10	O	0,013	0,015	0,014	0,016	0,017	0,019	0,014	0,016	0,013	0,014
8	10	F	0,009	0,01	0,009	0,01	0,009	0,01	0,01	0,01	0,01	0,009
8	10	O	0,016	0,016	0,017	0,016	0,016	0,017	0,017	0,016	0,016	0,016
12	10	F	0,006	0,007	0,006	0,007	0,007	0,006	0,007	0,008	0,006	0,007
12	10	O	0,018	0,01	0,011	0,01	0,012	0,01	0,011	0,01	0,01	0,011
17	10	F	0,006	0,006	0,007	0,007	0,008	0,007	0,007	0,008	0,006	0,007
17	10	O	0,008	0,008	0,009	0,008	0,018	0,012	0,011	0,012	0,01	0,011
21	10	F	0,005	0,004	0,005	0,004	0,005	0,006	0,004	0,005	0,005	0,005
21	10	O	0,009	0,01	0,009	0,01	0,011	0,011	0,011	0,011	0,012	0,011
30	10	F	0,004	0,005	0,005	0,004	0,004	0,005	0,005	0,004	0,004	0,004
30	10	O	0,007	0,007	0,007	0,008	0,008	0,008	0,007	0,008	0,008	0,007
32	10	F	0,006	0,006	0,007	0,007	0,006	0,006	0,006	0,008	0,007	0,007
32	10	O	0,011	0,012	0,013	0,011	0,011	0,013	0,012	0,012	0,014	0,013
36	10	F	0,008	0,009	0,008	0,01	0,009	0,009	0,008	0,009	0,009	0,008
36	10	O	0,013	0,013	0,014	0,013	0,014	0,013	0,014	0,018	0,014	0,013
40	10	F	0,009	0,01	0,009	0,01	0,01	0,01	0,009	0,01	0,011	0,01
40	10	O	0,015	0,017	0,018	0,016	0,015	0,018	0,017	0,016	0,017	0,016
43	10	F	0,005	0,005	0,006	0,006	0,009	0,006	0,006	0,006	0,007	0,006
43	10	O	0,01	0,011	0,01	0,012	0,013	0,013	0,013	0,013	0,012	0,013
45	10	F	0,004	0,005	0,005	0,005	0,005	0,006	0,005	0,005	0,005	0,005
45	10	O	0,005	0,007	0,006	0,005	0,005	0,005	0,006	0,005	0,005	0,005
50	10	F	0,009	0,009	0,008	0,01	0,009	0,009	0,009	0,009	0,01	0,01
50	10	O	0,012	0,013	0,014	0,013	0,014	0,014	0,013	0,014	0,013	0,013
6	11	F	0,009	0,009	0,008	0,009	0,009	0,009	0,009	0,009	0,009	0,009
6	11	O	0,012	0,013	0,012	0,013	0,024	0,015	0,022	0,013	0,016	0,012
11	11	F	0,008	0,009	0,008	0,009	0,009	0,009	0,008	0,009	0,009	0,008
11	11	O	0,012	0,013	0,012	0,013	0,014	0,012	0,013	0,012	0,012	0,012
16	11	F	0,006	0,006	0,006	0,006	0,007	0,006	0,006	0,006	0,008	0,007

16	11	O	0,01	0,01	0,011	0,011	0,011	0,012	0,01	0,012	0,02	0,015
0	12	F	0,006	0,008	0,007	0,008	0,007	0,007	0,007	0,008	0,007	0,006
0	12	O	0,005	0,006	0,006	0,005	0,006	0,007	0,006	0,005	0,006	0,007
1	12	F	0,01	0,009	0,008	0,01	0,01	0,01	0,009	0,009	0,01	0,01
1	12	O	0,013	0,015	0,016	0,021	0,016	0,013	0,014	0,014	0,014	0,013
33	12	F	0,008	0,01	0,009	0,009	0,009	0,009	0,009	0,009	0,01	0,01
33	12	O	0,013	0,013	0,013	0,013	0,013	0,014	0,015	0,015	0,013	0,012
48	12	F	0,005	0,005	0,004	0,004	0,005	0,005	0,005	0,004	0,004	0,005
48	12	O	0,007	0,008	0,007	0,007	0,007	0,009	0,007	0,007	0,008	0,007
10	13	F	0,004	0,004	0,005	0,005	0,005	0,005	0,004	0,004	0,004	0,004
10	13	O	0,006	0,006	0,006	0,006	0,005	0,006	0,006	0,006	0,006	0,007
23	13	F	0,009	0,01	0,01	0,011	0,01	0,01	0,01	0,01	0,011	0,01
23	13	O	0,015	0,016	0,018	0,017	0,015	0,016	0,016	0,017	0,019	0,017
29	14	F	0,005	0,005	0,005	0,004	0,004	0,009	0,005	0,007	0,015	0,009
29	14	O	0,006	0,007	0,009	0,007	0,007	0,006	0,006	0,008	0,007	0,007
14	15	F	0,005	0,005	0,005	0,004	0,004	0,006	0,005	0,005	0,005	0,005
14	15	O	0,011	0,011	0,011	0,012	0,011	0,012	0,011	0,016	0,017	0,017
26	15	F	0,008	0,008	0,008	0,01	0,009	0,009	0,01	0,009	0,009	0,009
26	15	O	0,018	0,018	0,022	0,018	0,018	0,019	0,018	0,019	0,02	0,02
39	15	F	0,007	0,007	0,007	0,007	0,008	0,007	0,007	0,008	0,007	0,007
39	15	O	0,012	0,013	0,014	0,012	0,013	0,013	0,012	0,013	0,012	0,017
27	16	F	0,006	0,008	0,006	0,006	0,007	0,007	0,006	0,007	0,006	0,006
27	16	O	0,007	0,007	0,007	0,007	0,008	0,007	0,007	0,007	0,007	0,007
3	17	F	0,01	0,012	0,01	0,011	0,01	0,011	0,011	0,012	0,011	0,012
3	17	O	0,019	0,019	0,019	0,019	0,019	0,021	0,021	0,019	0,019	0,02
41	17	F	0,005	0,005	0,005	0,005	0,005	0,007	0,005	0,005	0,005	0,005
41	17	O	0,012	0,013	0,014	0,014	0,014	0,013	0,014	0,018	0,016	0,016
46	18	F	0,011	0,012	0,011	0,011	0,012	0,011	0,012	0,011	0,011	0,012
46	18	O	0,027	0,029	0,027	0,027	0,027	0,027	0,028	0,028	0,028	0,028
44	20	F	0,013	0,015	0,014	0,014	0,014	0,013	0,014	0,013	0,014	0,013
44	20	O	0,05	0,051	0,051	0,051	0,051	0,052	0,053	0,051	0,053	0,051
47	21	F	0,005	0,006	0,006	0,005	0,006	0,006	0,006	0,006	0,006	0,007
47	21	O	0,012	0,013	0,013	0,013	0,014	0,013	0,014	0,013	0,013	0,014
22	22	F	0,016	0,018	0,017	0,017	0,018	0,016	0,017	0,016	0,016	0,017
22	22	O	0,065	0,064	0,044	0,027	0,028	0,028	0,028	0,028	0,028	0,028
51	22	F	0,01	0,01	0,01	0,01	0,01	0,013	0,009	0,01	0,011	0,018
51	22	O	0,018	0,015	0,016	0,016	0,016	0,016	0,016	0,016	0,016	0,016
15	24	F	0,015	0,015	0,014	0,015	0,018	0,014	0,015	0,015	0,015	0,015
15	24	O	0,042	0,043	0,043	0,043	0,043	0,043	0,042	0,043	0,042	0,043
42	24	F	0,006	0,007	0,007	0,007	0,008	0,007	0,007	0,006	0,008	0,009
42	24	O	0,022	0,022	0,023	0,028	0,026	0,027	0,027	0,027	0,031	0,032
49	24	F	0,014	0,015	0,015	0,014	0,016	0,015	0,015	0,015	0,015	0,015
49	24	O	0,042	0,043	0,043	0,043	0,047	0,042	0,043	0,044	0,043	0,043
20	25	F	0,095	0,096	0,104	0,119	0,141	0,141	0,17	0,106	0,105	0,119
20	25	O	0,344	0,395	0,417	0,407	0,304	0,299	0,437	0,413	0,394	0,269
38	25	F	0,166	0,096	0,136	0,222	0,102	0,096	0,095	0,096	0,098	0,142

38	25	O	0,276	0,268	0,269	0,369	0,366	0,275	0,533	0,292	0,443	0,269
4	29	F	0,008	0,013	0,013	0,012	0,013	0,012	0,012	0,012	0,012	0,012
4	29	O	0,159	0,16	0,078	0,069	0,073	0,066	0,075	0,096	0,14	0,169
28	29	F	0,006	0,007	0,007	0,007	0,008	0,007	0,007	0,006	0,007	0,007
28	29	O	0,023	0,023	0,024	0,024	0,024	0,024	0,024	0,023	0,024	0,024
7	30	F	0,007	0,007	0,008	0,007	0,008	0,007	0,007	0,009	0,008	0,008
7	30	O	0,03	0,033	0,031	0,031	0,031	0,034	0,037	0,037	0,039	0,038
24	31	F	0,007	0,007	0,007	0,007	0,008	0,007	0,007	0,007	0,008	0,009
24	31	O	0,036	0,042	0,053	0,053	0,07	0,071	0,071	0,071	0,071	0,07
34	32	F	0,017	0,017	0,017	0,017	0,016	0,017	0,017	0,018	0,017	0,016
34	32	O	0,05	0,051	0,052	0,051	0,051	0,051	0,051	0,051	0,053	0,051
9	35	F	0,041	0,041	0,04	0,042	0,04	0,042	0,042	0,04	0,042	0,041
9	35	O	0,087	0,047	0,047	0,047	0,047	0,048	0,047	0,05	0,047	0,047
19	37	F	0,022	0,022	0,022	0,022	0,025	0,024	0,022	0,022	0,022	0,024
19	37	O	0,215	0,125	0,088	0,089	0,09	0,097	0,088	0,088	0,089	0,088
35	37	F	0,018	0,018	0,018	0,018	0,019	0,018	0,018	0,018	0,02	0,018
35	37	O	0,079	0,08	0,079	0,07	0,064	0,082	0,095	0,084	0,08	0,066
5	38	F	0,296	0,119	0,129	0,12	0,12	0,119	0,122	0,126	0,12	0,119
5	38	O	0,431	0,587	0,339	0,337	0,467	0,513	0,465	0,478	0,455	0,398
25	56	F	0,304	0,255	0,198	0,205	0,198	0,255	0,222	0,227	0,229	0,221
25	56	O	2,276	2,122	2,111	2,272	2,137	2,105	2,1	2,24	2,796	2,235
37	57	F	0,07	0,07	0,07	0,071	0,07	0,073	0,071	0,071	0,07	0,07
37	57	O	0,236	0,24	0,117	0,107	0,143	0,152	0,096	0,096	0,096	0,096
18	59	F	0,032	0,033	0,032	0,032	0,054	0,042	0,043	0,045	0,042	0,043
18	59	O	0,277	0,125	0,125	0,126	0,128	0,126	0,125	0,125	0,126	0,126
31	63	F	0,049	0,048	0,049	0,049	0,049	0,049	0,051	0,049	0,049	0,049
31	63	O	3,277	3,06	3,702	3,49	3,748	3,055	3,211	3,033	3,202	3,204
13	137	F	0,082	0,083	0,084	0,083	0,123	0,124	0,104	0,083	0,105	0,124
13	137	O	0,765	0,761	0,766	0,588	0,588	0,761	0,765	0,59	0,59	0,603

Tabela A.2: Média dos tempos de execução Funsat e OBDD

Num.	*Features*	Restrições	$\mu_{Funsat(s)}$	$\mu_{OBDD(s)}$
0	10	15	0,0048	0.01050
10	10	17	0,0089	0.01510
15	10	16	0,0096	0.01630
21	10	15	0,0066	0.01220
22	10	15	0,0069	0.01070
23	10	14	0,0092	0.01330
27	10	15	0,0098	0.01650
29	10	16	0,0044	0.00750
30	10	19	0,0087	0.01390
43	10	16	0,0062	0.01200
48	10	15	0,0067	0.01130
50	10	16	0,005	0.00540

28	11	19	0,0064	0.01220
37	11	16	0,0086	0.01250
42	11	20	0,0089	0.01520
2	12	16	0,0092	0.01340
5	12	18	0,0095	0.01490
8	12	19	0,0046	0.00740
20	12	17	0,0071	0.00590
4	13	21	0,0101	0.01660
19	13	14	0,0044	0.00600
17	14	21	0,0068	0.00700
9	15	27	0,0049	0.01290
40	15	29	0,0072	0.01310
44	15	22	0,0089	0.01900
49	16	18	0,0065	0.00710
11	17	27	0,0052	0.01440
16	17	27	0,011	0.01950
46	18	34	0,0114	0.02760
6	20	40	0,0137	0.05140
12	21	34	0,0059	0.01320
24	22	39	0,0168	0.03680
41	22	34	0,0111	0.01610
18	24	43	0,0151	0.04270
32	24	43	0,0072	0.02650
45	24	43	0,0149	0.04330
1	25	34	0,1249	0.33600
33	25	34	0,1196	0.36790
3	29	40	0,0119	0.10850
34	29	40	0,0069	0.02370
31	30	56	0,0076	0.03410
36	31	44	0,0074	0.06080
38	32	43	0,0169	0.05120
39	35	59	0,0411	0.05140
14	37	61	0,0183	0.07790
25	37	58	0,0227	0.10570
47	38	65	0,139	0.44700
26	56	169	0,2314	2.23940
51	57	79	0,0706	0.13790
35	59	87	0,0398	0.14090
7	63	129	0,0491	3.29820
13	137	179	0,0995	0.67770

Tabela A.3: Geração de Número de Produtos de *Features*

Modelo	*Features*	Restrições	$\mu_{Tempo(s)}$	Número de Produtos
2	10	17	0,0145	9
8	10	16	0,0169	16

12	10	15	0,0112	24
17	10	15	0,0116	31
21	10	15	0,017	17
30	10	16	0,0095	16
32	10	15	0,0122	16
36	10	19	0,0138	1
40	10	15	0,0165	20
43	10	16	0,0169	16
45	10	16	0,0055	12
50	10	14	0,0132	48
6	11	20	0,0129	2
11	11	16	0,0129	35
16	11	19	0,0141	4
0	12	17	0,008	64
1	12	18	0,014	200
33	12	16	0,0131	240
48	12	19	0,0074	28
10	13	14	0,007	624
23	13	21	0,0164	12
29	14	21	0,0071	64
14	15	27	0,0222	4
26	15	22	0,0121	50
39	15	29	0,018	4
27	16	18	0,0087	3645
3	17	27	0,0201	96
41	17	27	0,0167	112
46	18	34	0,0286	8
44	20	40	0,0526	132
47	21	34	0,017	128
22	22	39	0,0313	63
51	22	34	0,027	1216
15	24	43	0,045	16
42	24	43	0,0452	16
49	24	43	0,0451	16
20	25	34	0,3454	1620
38	25	34	0,3729	1620
4	29	40	0,1084	129024
28	29	40	0,0249	197376
7	30	56	0,0767	14
24	31	44	0,0402	319488
34	32	43	0,0543	33554432
9	35	59	0,0612	1152
19	37	58	0,1265	1316772
35	37	61	0,0981	1296
5	38	65	0,413	4920
25	56	169	2,2092	128

37	57	79	0,1438	8776581120
18	59	87	0,1617	95551488
31	63	129	3,0873	1683117600
13	137	179	0,7199	1,56939401208204E+23

Tabela A.4: Grau de Homogeneidade dos Modelos de *Features*

Modelos	*Features*	Restrições	$\mu_{Tempo(s)}$	Homogeneidade
2	10	17	0,0507	1
8	10	16	0,0889	1
12	10	15	0,0347	1
17	10	15	0,0362	1
21	10	15	0,0995	1
30	10	16	0,0466	1
32	10	15	0,0356	1
36	10	19	0,047	-9
40	10	15	0,0496	1
43	10	16	0,0653	1
45	10	16	0,0185	1
50	10	14	0,041	1
6	11	20	0,0368	1
11	11	16	0,0535	1
16	11	19	0,0516	1
0	12	17	0,0288	1
1	12	18	0,069	1
33	12	16	0,0429	1
48	12	19	0,0343	1
10	13	14	0,0442	1
23	13	21	0,0763	1
29	14	21	0,0257	1
14	15	27	0,1471	1
26	15	22	0,0956	1
39	15	29	0,1185	1
27	16	18	0,055	1
3	17	27	0,1173	1
41	17	27	0,1186	1
46	18	34	0,1521	1
44	20	40	0,2849	1
47	21	34	0,1783	1
22	22	39	0,4748	1
51	22	34	0,2287	1
15	24	43	0,2792	1
42	24	43	0,283	1
49	24	43	0,3213	1
20	25	34	8,0811	1

38	25	34	7,6712	1
4	29	40	1,4588	1
28	29	40	0,3798	1
7	30	56	0,6781	1
24	31	44	0,6438	1
34	32	43	0,3743	1
9	35	59	1,4985	1
19	37	58	2,7365	1
35	37	61	2,0036	1
5	38	65	13,8813	1
25	56	169	100,6588	1
37	57	79	4,864	1
18	59	87	5,8971	1
31	63	129	146,7759	1
13	137	179	72,9357	1

Tabela A.5: Fator de Variabilidade dos Modelos de *Features*

Modelos	*Features*	Restrições	$\mu_{Tempo(s)}$	Fator de Variabilidade
2	10	17	0,0148	0,00879
8	10	16	0,0079	0,01563
12	10	15	0,0118	0,02344
17	10	15	0,0115	0,03027
21	10	15	0,0099	0,0166
30	10	16	0,0126	0,01563
32	10	15	0,0123	0,01563
36	10	19	0,0139	0,00098
40	10	15	0,0077	0,01953
43	10	16	0,0172	0,01563
45	10	16	0,0103	0,01172
50	10	14	0,0139	0,04688
6	11	20	0,0134	0,00098
11	11	16	0,0134	0,01709
16	11	19	0,0064	0,00195
0	12	17	0,0099	0,01563
1	12	18	0,0146	0,04883
33	12	16	0,0137	0,05859
48	12	19	0,0072	0,00684
10	13	14	0,0137	0,07617
23	13	21	0,0168	0,00146
29	14	21	0,0119	0,00391
14	15	27	0,0103	0,00012
26	15	22	0,0113	0,00153
39	15	29	0,0132	0,00012
27	16	18	0,0129	0,05562

3	17	27	0,0131	0,00073
41	17	27	0,0211	0,00085
46	18	34	0,0126	3E-05
44	20	40	0,0279	0,00013
47	21	34	0,031	6E-05
22	22	39	0,0294	2E-05
51	22	34	0,0237	0,00029
15	24	43	0,0198	0
42	24	43	0,022	0
49	24	43	0,0201	0
20	25	34	0,3976	5E-05
38	25	34	0,3783	5E-05
4	29	40	0,0818	0,00024
28	29	40	0,025	0,00037
7	30	56	0,0352	0
24	31	44	0,0312	0,00015
34	32	43	0,0235	0,00781
9	35	59	0,0938	0
19	37	58	0,0906	1E-05
35	37	61	0,1154	0
5	38	65	0,4592	0
25	56	169	2,2894	0
37	57	79	0,1627	0
18	59	87	0,1597	0
31	63	129	3,0742	0
13	137	179	0,6781	0

Tabela A.6: Tempo de Execução na Análise de Modelo de *Features*

Modelos	*Features*	Restrições	$\mu_{TempodeExecuo(s)}$
0	10	15	0,0124
14	10	17	0,004
22	10	16	0,0104
32	10	15	0,0051
37	10	15	0,0055
38	10	14	0,0041
40	10	19	0,0058
43	10	16	0,0068
44	10	15	0,0066
49	10	16	0,0066
50	10	16	0,0047
51	10	19	0,005
54	10	16	0,0103
56	10	16	0,0065
70	10	14	0,0039

79	10	16	0,0106
85	10	15	0,0036
89	10	12	0,0073
90	10	16	0,0076
46	11	19	0,0048
53	11	15	0,0077
66	11	16	0,0049
67	11	17	0,0036
77	11	20	0,0078
3	12	16	0,0077
4	12	19	0,0095
8	12	18	0,0078
9	12	18	0,0049
12	12	19	0,0038
31	12	17	0,0076
6	13	21	0,0037
30	13	14	0,0039
87	13	20	0,0079
16	14	23	0,0068
26	14	21	0,0046
45	14	18	0,0088
13	15	27	0,004
72	15	29	0,0143
80	15	22	0,0081
52	16	16	0,0089
88	16	18	0,009
17	17	27	0,009
23	17	27	0,0044
57	18	21	0,0093
73	18	32	0,0124
82	18	34	0,0068
1	19	33	0,0292
7	20	29	0,0095
10	20	40	0,0258
28	20	32	0,0118
69	20	38	0,0099
18	21	34	0,0099
33	22	30	0,004
39	22	39	0,0398
74	22	34	0,0042
27	24	43	0,0104
29	24	43	0,0104
47	24	43	0,0049
59	24	43	0,0105
81	24	43	0,0104
2	25	34	0,0092

36	25	38	0,0119
55	25	48	0,0048
60	25	34	0,0096
61	25	41	0,0051
75	27	55	2,2848
62	28	57	0,2075
5	29	40	0,0223
24	29	35	0,0199
63	29	40	0,0142
76	29	53	0,0351
84	29	44	0,8437
86	29	39	0,0107
58	30	56	0,0137
25	31	48	0,0296
65	31	44	0,0145
78	31	50	0,0306
68	32	43	0,015
34	34	56	0,0115
35	34	46	0,0105
71	35	59	0,0051
21	37	61	0,0115
41	37	58	0,0645
83	38	65	0,9902
19	41	53	0,0102
42	56	169	14,8169
91	57	79	0,0139
64	59	87	0,0434
48	60	82	0,0346
11	63	129	0,4174
15	88	160	792,7781
20	137	179	0,0891

Referências

[1] Charles W. Krueger. *New Methods in Software Product Line Practice Examining the benefits of next-generation SPL methods, Article, Communications of the ACM, Vol 49, December,*2006. vii, 1, 10, 11

[2] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns, Book, Addison-Wesley,* 2001. 1, 5

[3] Klaus Pohl, Gunter Bockle and Frank Van Der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques, Book, Springer, 2005.* 1, 4, 6, 8

[4] David L. Parnas. *On the Design and Development of Program Families, Article, IEEE Transactions on Software Engineering, Vol. SE-2,* 1975. 1

[5] ConSiste - Systems Conception Laboratory. DIMAp - Departament of Informatics and Applied Mathematics *http://www.dimap.ufrn.br/lightplacmestudio/gingaforall.php.* vii, 2

[6] Rodrigo Bonifácio and Leopoldo Teixeira and Paulo Borba. *Hephaestus A Tool for Managing SPL Variabilities, Article, Informatics Center, Federal University of Pernambuco,* (UFPE), 2009. vii, 2, 12, 14

[7] David Benavides, Sergio Segura and Antonio Ruiz-Cortés. *Automated Analysis of Feature Models 20 Years Later: A Literature Review, Article, Dpt. de Lenguajes y Sistemas Informáticos, University of Seville, 2006.* vii, 2, 16, 17, 18, 20, 21, 22, 23, 26, 40

[8] Linha de Produção *http://pt.wikipedia.org/wiki/Linhadeproducao, Site, Wikipédia.* 4

[9] Customização em Massa *http://pt.wikipedia.org/wiki/Customizaçãoemmassa, Site, Wikipédia.* 4

[10] *SearchServerVirtualization http://searchservervirtualization.techtarget.com /definition/platform, Site, SearchServerViualization.* 4

[11] Frank van der Linden,Klaus Schmid and Eelco Rommes. *Software Product Lines In Action - The Best Industrial Practice In Product Line Engineering, Book, Springer, 2007.* vii, 4, 5, 6

[12] Roberto C. Durscki, Mauro M. Spinola, Robert C. Burnett and Sheila S. Reinehr. Linha de Produto de Software: riscos e vantangens de sua implantação, Artigo, 2004. 6

[13] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak and A. Spencer Peterson. *Feature-Oriented Domain Analysis (FODA), Tecnical Report, CMU, SEI-90-TR-21, ESD-90-TR-222, November, 1990.* 8

[14] Leopoldo Teixeira, Paulo Borba and Rohit Ghevi. *Safe Composition of Configuration Knowledge-based Software Product Lines, Informatics Center, Article, Federal University of Pernambuco, University of Campina Grande, 2011.* 11

[15] Catal Cagaty and Diri Banu *A Conceptual Framework to Integrate Fault Prediction Sub-process for Software Product Lines, Article, TUBITAK, Marmara Research Center, Information Technologies Institute, Kocaeli, Turkey, Yildiz Techinical University, Department of Computer Engineering, Istambul, Turkey, 2008.* 6

[16] Lucinéia Turnes, Rodrigo Bonifácio,Vander Alves and Ralf Lammel. *Techniques for Developing a Product Line of Product Line Tools: a Comparative Study, Article, Computer Science Department, University of Brasilia , Software Languages Team, University Koblenz-Landau, Germany,2011.* vii, 13

[17] Rodrigo Bonifácio de Almeida. *Modeling Software Product Line Variability in Use Case Scenarios, An Approach Basead on Crosscutting Mechanisms, Informatics Center, Thesis, Federal University of Pernambuco, (UFPE), 2010.* vii, 12, 13, 15

[18] David Benavides, Pablo Trinidad and Antonio Ruiz-Cortés. *Automated Reasoning on Features Models, Article, 2005, Dpto. de Lenguajes y Systemas Informáticos, University of Seville.* vii, 8, 9, 10

[19] Krzysztof Czarnecki, Simon Helsen and Ulrich Eisenecker. *Staged Configuration Using Feature Models, Article, University of Waterloo, Canada, University of Applied Sciences Kaiserslautern, Zweibrucken, Germany, 2004.* 9

[20] Rodrigo Bonifacio. *FCTypeChecker.lhs, Literate Haskell, hephaestus-hephaestus, feature-modeling, src, FeatureModel, 2008-2009.* vii, 15

[21] John E. Freund Estatística Aplicada - Economia, Administração e Contabilidade, 11ª edição, Editora Bookman. 32