



UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

JOÃO HOLANDA FREIRES JÚNIOR

**DyMMer: UMA FERRAMENTA PARA AVALIAÇÃO DE QUALIDADE DO
MODELO DE *FEATURES* BASEADA EM MEDIDAS DE LINHAS DE PRODUTOS
DE SOFTWARE DINÂMICAS**

**QUIXADÁ
2014**

JOÃO HOLANDA FREIRES JÚNIOR

**DyMMEr: UMA FERRAMENTA PARA AVALIAÇÃO DE QUALIDADE DO
MODELO DE *FEATURES* BASEADA EM MEDIDAS DE LINHAS DE PRODUTOS
DE SOFTWARE DINÂMICAS**

Trabalho de Conclusão de Curso
submetido à Coordenação do Curso
Bacharelado em Sistemas de Informação
da Universidade Federal do Ceará como
requisito parcial para obtenção do grau de
Bacharel.

Área de concentração: Computação

Orientadora Prof^a. Msc. Carla Ilane
Moreira Bezerra

**QUIXADÁ
2014**

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca do Campus de Quixadá

F933d Freires Júnior, João Holanda
DyMMER: uma ferramenta para avaliação de qualidade do modelo de linhas de produtos de software dinâmicas / Suelhy Alves Costa.– 2014.
66 f. : il. color., enc. ; 30 cm.

Monografia (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Sistemas de Informação, Quixadá, 2014.
Orientação: Prof. Me. Carla Ilane Moreira Bezerra
Área de concentração: Computação

1. Software – Controle de qualidade 2. Engenharia de software 3. Sistemas de computação I.
Título.

CDD 005.1

JOÃO HOLANDA FREIRES JÚNIOR

**UMA FERRAMENTA PARA AVALIAÇÃO DE QUALIDADE DO MODELO DE
FEATURES BASEADA EM MEDIDAS DE LINHAS DE PRODUTOS DE
SOFTWARE DINÂMICAS**

Trabalho de Conclusão de Curso
submetido à Coordenação do Curso
Bacharelado em Sistemas de Informação
da Universidade Federal do Ceará como
requisito parcial para obtenção do grau de
Bacharel.

Área de concentração: Computação

Aprovado em: 01/ dezembro / 2014.

BANCA EXAMINADORA

Profa. Msc. Carla Ilane Moreira Bezerra
Universidade Federal do Ceará-UFC

Profa. Dra. Rossana Maria de Castro Andrade
Universidade Federal do Ceará-UFC

Prof. Dr. José Maria da Silva Monteiro Filho
Universidade Federal do Ceará-UFC

Profa. Dra. Tânia Saraiva de Melo Pinheiro
Universidade Federal do Ceará-UFC

AGRADECIMENTOS

Agradeço primeiramente à Deus, pelas oportunidades a mim dadas e todo seu conforto para que eu pudesse ter forças e fé para conquistá-las.

Sou grato pela educação e todos os esforços de minha família, em especial aos meus pais, Aila Maria e João Holanda, e a minha irmã, Karina Holanda.

Todo qualquer agradecimento à minha grande companheira, Luana Montenegro, que esteve sempre comigo, apoiando-me nos desafios e me confortando em todas as dificuldades, dando-me alegria e todo seu carinho.

À Prof^a. Msc. Carla Ilane Moreira Bezerra, meus agradecimentos pela oportunidade de seu trabalho e a excelente qualidade em sua orientação.

Agradeço aos professores que participaram da banca, Profa. Dra. Rossana Maria de Castro Andrade, Prof. Dr. José Maria da Silva Monteiro Filho e Profa. Dra. Tânia Saraiva de Melo Pinheiro por disponibilizarem seu tempo para apreciar e contribuir na construção deste trabalho.

E agradeço também aos amigos que, presentes ou distantes, ajudaram-me de alguma forma, seja em palavras ou atitudes.

“Talvez não tenha conseguido fazer o melhor, mas lutei para que o melhor fosse feito.
Não sou o que deveria ser, mas Graças a Deus, não sou o que era antes”.

(Marthin Luther King)

RESUMO

Linhas de Produto de Software (LPS) é uma abordagem que tem como ponto central a reutilização sistemática de artefatos de software baseados em uma família de produtos, concebidos desde o início com o propósito de reuso. No entanto, as LPSs representam as variabilidades dos produtos gerados apenas de forma estática, ou seja, a adaptação ocorre apenas em tempo de desenvolvimento, necessitando que sejam geradas diferentes configurações conforme a necessidade do cliente e seu ambiente de execução. Surgem então as Linhas de Produtos de Software Dinâmicas (LPSD), permitindo o desenvolvimento de softwares capazes de se adaptar aos diferentes requisitos e restrições diante das mudanças correntes do ambiente. É importante garantir a qualidade dos artefatos e produtos de software a serem fornecidos pelas LPSs e LPSDs. Uma abordagem mais viável para garantir a qualidade da LPS seria avaliar a qualidade nas fases iniciais de desenvolvimento da LPS, prevenindo assim que erros se propaguem para outras fases de desenvolvimento e consequentemente nos produtos finais. Aplicar medidas de qualidade nos artefatos pode prevenir que erros sejam passados para fases posteriores do desenvolvimento. Este trabalho tem como objetivo desenvolver uma ferramenta para avaliar a qualidade do modelo de *features* a fim de inserir outras medidas como novas funcionalidades do sistema para avaliar atributos de qualidade dos modelos de *features* de LPSs e LPSDs. Também foram definidas na ferramenta funcionalidades para edição dos modelos de *features* para inserção de adaptações de contexto e incluindo regras de adaptação de contexto para ativação e desativação de *features*. Para avaliação da ferramenta foram utilizados trinta modelos de *features*, aplicando-se todas as medidas e exportando seus resultados, conforme funcionalidade implementada na ferramenta, para serem analisadas pelo *software* Epi Info®.

Palavras chave: Medidas. Avaliação de Qualidade. Modelo de *Features*, Linha de Produtos de Software Dinâmicas.

ABSTRACT

Software Product Lines (SPL) is an approach that has as its central point the systematic reuse of software artifacts based on a family of products designed from the outset for the purpose of reuse. However, the SPLs represent the variability of products generated only in a static way, ie adaptation occurs only at design time, requiring different configurations are generated as needed by the customer and its execution environment. Then come the Dynamic Software Product Lines (DSPL), allowing the development of software capable of adapting to different requirements and restrictions on the current changes in the environment. It is important to ensure the quality of artifacts and software products to be supplied by the SPLs and DSPLs. A more viable approach to ensure the quality of SPL would evaluate quality in the early stages of development of SPL, thereby preventing errors from propagating to other stages of development and consequently the final products. Apply quality measures in artifacts can prevent errors are passed to later stages of development. This work aims to develop a tool to assess the quality of the feature model in order to include other measures such as new features of the system to evaluate quality attributes of feature models of SPLs and DSPLs. It were also defined in the tool features for editing the feature models for inserting context of adjustments and including context of adaptation rules for activation and deactivation features. For evaluation tool was used thirty feature models, applying all measures and exporting its results, as implemented in the tool functionality, to be analyzed by Epi Info® software.

Keywords: Measures. Quality Assessment. Features Model, Dynamic Software Product Line.

LISTA DE QUADROS

Tabela 1: Comparação entre notações.....	21
Tabela 2: Características e atributos de qualidade das medidas identificadas para avaliação da qualidade do modelo de <i>features</i>	27
Tabela 3: Conjunto de medidas identificadas para avaliar a qualidade do modelo de <i>features</i>	28
Tabela 4: Continuação da Tabela 3.....	29
Tabela 5: Recomendação de aplicação para SAT e BDD.....	42

LISTA DE ILUSTRAÇÕES

Figura 1: Engenharia de Linha de Produtos de Software.....	14
Figura 2: (a) Visão geral de uma Aplicação Tradicional e (b) uma Aplicação Sensível ao contexto.....	16
Figura 3: Mecanismos de variabilidade em tempo de execução necessários para a transição de LPSs em LPSDs.....	18
Figura 4: Exemplos de <i>features</i> mortas.....	22
Figura 5: Editor de modelos de <i>features</i>	30
Figura 6: Análise automatizada.....	31
Figura 7: Configuração de produto.....	32
Figura 8: Modelo de <i>features</i> do repositório.....	33
Figura 9: Operações de criação e análise.....	33
Figura 10: Operações de análise estrutural do modelo.....	34
Figura 11: Operações de configuração de produto.....	34
Figura 12: Operações de análise automatizada.....	35
Figura 13: Visão geral da interface gráfica do FAMILIAR.....	35
Figura 14: Procedimentos que serão realizados durante a execução deste trabalho.....	36
Figura 15: Imagem do diagrama de classes.....	40
Figura 16: Arquitetura da Ferramenta.....	41
Figura 17: S.P.L.A.R.....	41
Figura 18: Exemplo de três medidas implementadas.....	43
Figura 19: Estrutura representada pelo XML do S.P.L.O.T.....	44
Figura 20: Representação da estrutura do modelo.....	45
Figura 21: Componente de representação do modelo com contextos.....	46
Figura 22: Área de visualização.....	47
Figura 23: Seções da Área de edição.....	48
Figura 24: Ativar ou desativar <i>features</i>	49
Figura 25: Imagens do Excel com os dados exportados.....	50
Figura 26: Modelo de <i>features</i> sem adaptações de contexto.....	51
Figura 27: Validação exportação de modelo sem contexto.....	51
Figura 28: Modelo de <i>features</i> com adaptações de contexto.....	52
Figura 29: Validação exportação de modelo com contexto.....	53
Figura 30: Quantidade de <i>features</i> dos modelos selecionados do SPLOT.....	54
Figura 31: Análise com Número de <i>Features</i> x Complexidade Cognitiva.....	55
Figura 32: Análise com Profundidade da árvore de modelo x Complexidade Cognitiva.....	55
Figura 33: Análise com Número de <i>Features</i> x Flexibilidade de Configuração.....	56
Figura 34: Análise com Complexidade Cognitiva x Flexibilidade de Configuração.....	57
Figura 35: Análise com Número de <i>features</i> folha x Profundidade da Árvore.....	57
Figura 36: Análise com <i>Features</i> Variantes x Extensibilidade de <i>Features</i>	58

SUMÁRIO

1 INTRODUÇÃO.....	11
2 REVISÃO BIBLIOGRÁFICA.....	13
2.1 Linhas de Produtos de Software.....	13
2.2 Computação sensível ao contexto.....	15
2.3 Linhas de Produto de Software Dinâmicas.....	17
2.3.1 Variabilidade Dinâmica.....	17
2.3.2 Verificação e Ligação em Tempo de Execução.....	19
2.4 Modelos de <i>Features</i>	19
2.5 Qualidade em LPSs e LPSDs.....	23
2.5.1 Qualidade em LPSs.....	23
2.5.2 Qualidade em LPSDs.....	24
2.5.3 Medidas de Qualidade do Modelo de <i>Features</i>	26
2.6 Ferramentas para Avaliação da Qualidade do Modelo de <i>Features</i>	30
2.6.1 Ferramenta S.P.L.O.T.....	30
2.6.2 Familiar.....	33
3 PROCEDIMENTOS METODOLÓGICOS.....	36
3.1 Levantamento das Ferramentas Disponíveis para análise em modelos de <i>features</i>	36
3.2 Estudo da ferramenta S.P.L.O.T.....	36
3.3 Estudo da ferramenta FAMILIAR.....	37
3.4 Estudo das medidas que compõem a ferramenta.....	38
3.5 Modelagem da ferramenta.....	38
3.6 Implementação da ferramenta.....	38
3.7 Validação das medidas.....	39
4 PROJETO E DESENVOLVIMENTO DA FERRAMENTA.....	39
4.1.1 Requisitos funcionais.....	39
4.1.2 Arquitetura.....	40
4.2 Desenvolvimento.....	42
4.2.1 Importação.....	44
4.2.2 Área de visualização.....	46
4.2.3 Área de edição.....	47
4.2.4 Exportação.....	49
5 AVALIAÇÃO DO USO DA FERRAMENTA.....	50
5.1 Avaliação da Coleta das Medidas.....	50
5.2 Análise de Correlação das Medidas.....	53
6 CONSIDERAÇÕES FINAIS.....	58
REFERÊNCIAS.....	61

1 INTRODUÇÃO

Linhas de Produto de Software (LPS) é uma abordagem que tem como ponto central a reutilização sistemática de artefatos de software baseados em uma família de produtos, concebidos desde o início com o propósito de reuso. O objetivo de uma LPS é identificar características comuns e variáveis entre os possíveis produtos gerados, o que possibilita maior flexibilidade e adaptação conforme as necessidades do cliente, além da diminuição dos custos de produção e aumento da produtividade como fatores motivadores à sua implantação nas organizações (SILVA et al., 2010). Assim, no processo de engenharia de LPS, o passo inicial consiste em identificar necessidades similares e variabilidades dentro de um domínio de aplicação. A partir de similaridades e variabilidades encontradas no domínio, é possível derivar diferentes produtos a partir dessa linha.

Um artefato essencial nas LPSs é o modelo de *features*. Um modelo de *features* descreve as características identificadas nos possíveis produtos a serem gerados e as relações existentes entre as mesmas, representando todas as similaridades e variabilidades em uma LPS (BENAVIDES et al., 2010). Dessa forma, tal modelo consiste em um diagrama em formato de árvore, onde cada nó que representa possíveis *features* de sistemas que serão desenvolvidos na linha de produto. Uma *feature*, por sua vez, é uma característica comum e variável entre sistemas de um determinado domínio de aplicação, possibilitando a geração de diversos produtos com características diferentes. As *features* expressam similaridades e variabilidades de produtos de um domínio, podendo significar algo diferente para diferentes linhas de produto e domínio, tal como ser um requisito, uma funcionalidade ou um aspecto de qualidade (BEUCHE e DELGARNO, 2007).

No entanto, as LPSs representam as variabilidades dos produtos gerados apenas de forma estática, ou seja, a adaptação ocorre apenas em tempo de desenvolvimento, necessitando que sejam geradas diferentes configurações conforme a necessidade do cliente e seu ambiente de execução (FERNANDES et al. 2008). Contudo, com o advento de Aplicações Sensíveis ao Contexto, os softwares têm se tornado cada vez mais complexos, exigindo um alto grau de adaptação, dada as mudanças de requisitos e restrições conforme o ambiente a qual está inserido (HALLSTEINSEN et al., 2014). Surgem então as Linhas de Produtos de Software Dinâmicas (LPSD), permitindo o desenvolvimento de softwares capazes de se adaptar aos diferentes requisitos e restrições diante das mudanças correntes do

ambiente. Segundo HALLSTEINSEN et al. (2008), as LPDS podem ter as seguintes propriedades:

- Variabilidade dinâmica: configuração e vinculação em tempo de execução;
- Adaptar-se diversas vezes durante seu tempo de vida;
- Mudança de características variáveis durante sua execução (adição de *features* variáveis);
- Tratar-se com mudanças inesperadas;
- Tratar-se com mudanças de requisitos de usuário, como requisitos funcionais ou de qualidade;
- Sensível ao contexto (opcional); e
- Propriedades autoadaptativas (opcional).

Outro fator importante é a qualidade dos artefatos e produtos de software a serem fornecidos pelas LPSs e LPSDs. Entretanto, a avaliação da qualidade em uma LPS pode ser inviável caso se queira avaliar todos os produtos derivados da linha. Uma abordagem mais viável para garantir a qualidade da LPS seria avaliar a qualidade nas fases iniciais de desenvolvimento da LPS, prevenindo assim que erros se propaguem para outras fases de desenvolvimento e conseqüentemente nos produtos finais (ETXEBERRIA et al., 2008). O modelo de *features* é um artefato chave nas fases iniciais do desenvolvimento em uma LPS. Aplicar medidas de qualidade nesse artefato pode prevenir que erros sejam passados para fases posteriores do desenvolvimento. Existem diversas medidas relevantes a se considerar para a avaliação da qualidade de um modelo de *features*, medidas essas que se referem a uma característica de qualidade e afetam um atributo de qualidade respectivo (BEZERRA et al., 2013).

Existem poucas ferramentas que utilizam medidas para avaliação da qualidade do modelo de *features*, como por exemplo, o SPLOT (S.P.L.O.T, 2014) e o Familiar (FAMILIAR, 2014). Em ambas as ferramentas existem algumas medidas já definidas, porém apenas essas medidas podem não ser suficientes para avaliar a qualidade do modelo de *features*, pois as medidas presentes na análise automática da ferramenta não cobrem todos os atributos de qualidades necessários para a avaliação de um modelo de *features* (BEZERRA et al, 2013). Além disso, essas ferramentas não avaliam LPSDs, apenas LPSs tradicionais.

Neste contexto, este trabalho tem como objetivo desenvolver uma ferramenta para avaliar a qualidade do modelo de *features* a fim de inserir outras medidas como novas funcionalidades do sistema para avaliar atributos de qualidade dos modelos de *features* de LPSs e LPSDs. As medidas foram definidas no contexto de uma tese de doutorado. Parte das medidas foram publicadas em Bezerra et al. (2013) e complementadas em Bezerra et al. (2014). Uma grande inovação da ferramenta é a inserção de elementos de LPSDs como a ativação e desativação de *features* de acordo com o contexto (CAPILLA, 2014), permitindo a análise do modelo com as medidas de qualidade conforme tal contexto selecionado.

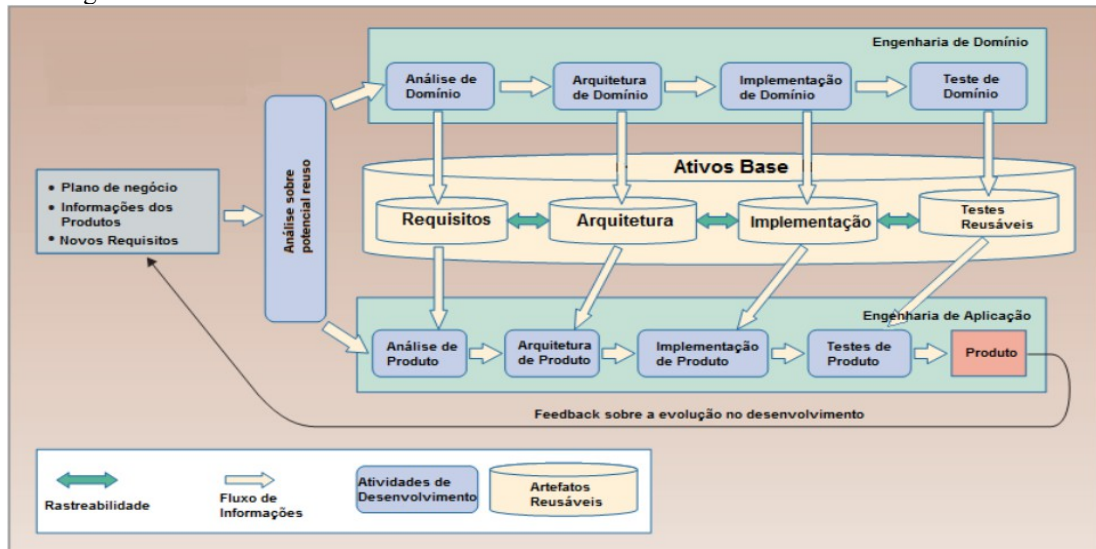
2 REVISÃO BIBLIOGRÁFICA

Neste Capítulo, serão apresentados os conceitos importantes utilizados no trabalho. Os conceitos estão relacionados à Linhas de Produtos de Software, Computação Sensível ao Contexto, Linhas de Produtos de Software Dinâmicas, Modelos de *Features*, Qualidade em LPSs e LPSDs e Ferramentas de avaliação de qualidade das LPSs.

2.1 Linhas de Produtos de Software

Uma Linha de Produtos de Software (LPS) tem como objetivo principal, o reuso planejado e sistemático de artefatos. A partir desses artefatos, podem-se derivar diversos produtos com similaridades e variabilidades entre si. Uma LPS pode promover uma maior produtividade e menores custos por que tem o reuso como ponto principal de sua abordagem. O processo de engenharia de linhas de produto de software tem duas fases, a engenharia de domínio e a engenharia de aplicação (SILVA et al. 2010). Na Figura 1, são apresentadas as atividades dessas fases.

Figura 1: Engenharia de Linha de Produtos de Software



Fonte: SILVA et al. 2010.

A entrada para as duas fases da engenharia de domínio e aplicação da LPS são planos de negócios, informações dos produtos e requisitos dos novos produtos a serem produzidos a partir da linha. O próximo passo é a análise sobre o potencial reuso, onde é analisado como poderá ser implementado o reuso na LPS. A partir disso entra-se nas duas fases principais da linha. A engenharia de domínio e a engenharia de aplicação (FERNANDES, 2009).

Durante a engenharia de domínio é definido o domínio a ser atendido pela linha de produto, e são identificadas as similaridades e variabilidades dentre os produtos a serem derivados para aquele domínio. Um artefato que expressa essas características que podem ser comuns ou variáveis aos produtos da linha é o modelo de *features*, ele é produzido durante a engenharia de domínio, mais exatamente na análise de domínio (SILVA et al., 2010).

Conforme visto na Figura 1, a engenharia de domínio é composta de diversas fases, segundo Fernandes (2009):

- A fase de análise de domínio tem como objetivo estudar uma classe de problemas de um determinado domínio a fim de conhecê-lo. A partir desse estudo pode-se então extrair um conjunto de similaridades e diferenças que irão compor os produtos desse domínio, essas similaridades e diferenças são expressas no modelo de *features*;
- A fase de arquitetura de domínio consiste em projetar a arquitetura da LPS, definindo alternativas arquiteturais que se adequem e atendam ao problema a ser tratado por produtos gerados na LPS;

- Em seguida existe a fase de implementação do domínio, onde são produzidos os componentes reusáveis para os produtos da linha; e
- Por fim existe a fase de testes de domínio, onde serão produzidos os testes reutilizáveis para qualquer variação de produto da linha.

Segundo Fernandes (2009), a engenharia de domínio tem como saída uma série de artefatos chamados de ativos base, entre eles temos requisitos, arquitetura, implementação e testes reusáveis. Esses ativos-base serão usados como entrada para cada fase da engenharia de aplicação:

- Na fase de análise de produto, os requisitos produzidos serão utilizados, para que sejam selecionadas as características do produto a ser derivado da linha;
- Na fase de arquitetura de produto a arquitetura será utilizada para definir qual das alternativas arquiteturais definidas para produtos do domínio atendido pela linha anteriormente, será o mais adequado para o um produto em particular a ser produzido naquele momento;
- A implementação do produto é a próxima fase que utiliza os componentes reusáveis produzidos durante a implementação do domínio; e
- Por fim, a fase de testes de produto pode ser realizada, utilizando os testes reutilizáveis no produto derivado no fim do processo de engenharia de aplicação.

2.2 Computação sensível ao contexto

Segundo Kjeldskov e Skov (2004), computação sensível ao contexto pode ser definida como “uma habilidade da aplicação de se adaptar as circunstâncias das mudanças e responder de acordo ao contexto de uso”. Complementando, conforme Brown et al. (1997), define contexto como a localização do usuário, identidades de pessoas próximas ao usuário, o tempo do dia, temperatura, etc. Já segundo Dey (1998), enumera contexto como o estado emocional do usuário, localização e orientação, tempo e data, objetos e pessoas no ambiente inserido.

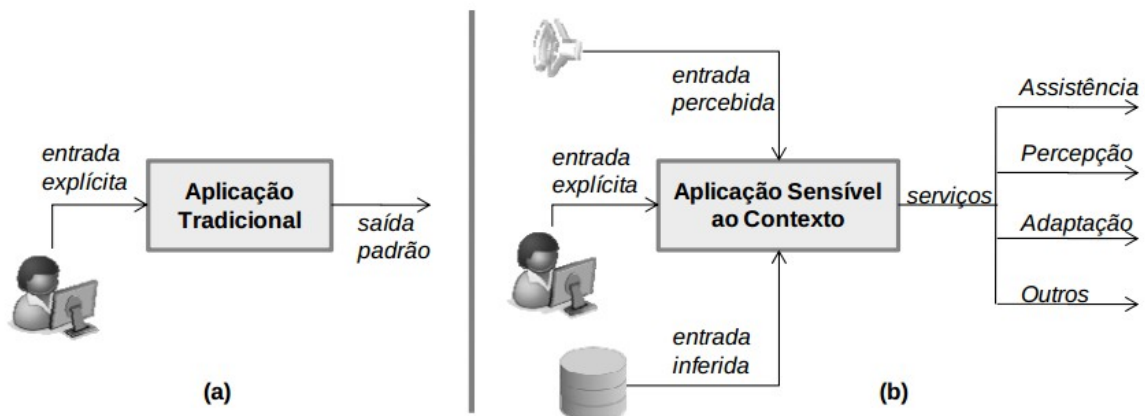
Sobretudo, o contexto como um todo é formado pelo ambiente que circunda a aplicação, possuindo um estado corrente, e as informações o qual envolve o estado do mesmo, sendo as informações de contexto. Dessa forma, uma definição amplamente aceita sobre informação de contexto, conforme Dey et al. (2001): “é qualquer informação que possa ser

utilizada para caracterizar uma entidade. Uma entidade é uma pessoa, lugar ou objeto considerado relevante para uma interação entre um usuário e uma aplicação, incluindo o usuário e a aplicação em questão”.

Assim, em sistemas computacionais, o contexto é uma importante ferramenta de apoio à comunicação entre os sistemas e seus usuários. Pois, compreendendo o contexto, o sistema pode adaptar-se e mudar suas sequências de ações, suas interações e o tipo de informação a ser fornecida para os usuários, conforme as alterações necessárias. Sobretudo, o contexto ainda auxilia os sistemas, em tempo de execução, habilitar ou desabilitar serviços e funcionalidades (ABOWD, 1999).

Segundo Dey e Abowd (2000), os sistemas sensíveis ao contexto são os que “utilizam o contexto para fornecer informações e/ou serviços relevantes para o usuário, onde relevância depende da tarefa do usuário”. Assim, conforme a, as aplicações sensíveis ao contexto consideram informações fornecidas pelos usuários, as armazenadas em uma base de conhecimento contextual, aquelas inferidas por meio de raciocínio e/ou aquelas percebidas a partir do ambiente, diferentemente das aplicações convencionais, as quais levam em conta apenas informações fornecidas explicitamente pelo usuário (VIEIRA et al., 2006).

Figura 2: (a) Visão geral de uma Aplicação Tradicional e (b) uma Aplicação Sensível ao contexto.



Fonte: VIEIRA et al. 2006.

Conforme a , os sistemas sensíveis ao contexto podem fornecer os seguintes serviços: assistência, percepção, adaptação, etc. Dentre eles, no serviço de assistência, o sistema pode dar suporte ao usuário visando à sua comodidade no ambiente em que está inserido, podendo aconselhar ou alertar sobre determinadas ações ou tarefas que se pode realizar para alcançar determinados objetivos, bem como fazer recomendações. Já no que se refere ao serviço de percepção, o sistema pode notificar ao usuário sobre pessoas e temas de seu interesse para um determinado contexto observado. Como seu terceiro serviço, a adaptação, refere-se a capacidade de autoadaptação do sistema, ou seja, modificar seus comportamentos e características conforme as mudanças percebidas no ambiente corrente ou às ações dos usuários (VIEIRA et al., 2006).

2.3 Linhas de Produto de Software Dinâmicas

Devido às aplicações sensíveis ao contexto serem emergentes, torna-se necessário que os sistemas sejam capazes de se adaptar dinamicamente conforme as mudanças e restrições do ambiente em que está. Tais sistemas foram projetados para se autoconfigurarem conforme um conjunto de aspectos em diferentes situações, bem como deve ser capaz de se adaptar às condições em tempo de execução com um mínimo de interferência humana. Contudo, as Linhas de Produto de Software convencionais não são capazes de prever aspectos dinâmicos, pois a adaptação de seus produtos é concebida durante o tempo de desenvolvimento (BENCOMO et al., 2012).

Dessa forma, surgem as Linhas de Produto de Software Dinâmicas (LPSD) como uma extensão das Linhas de Produto de Software convencionais, buscando identificar e tratar características em mudanças de software em tempo de execução.

2.3.1 Variabilidade Dinâmica

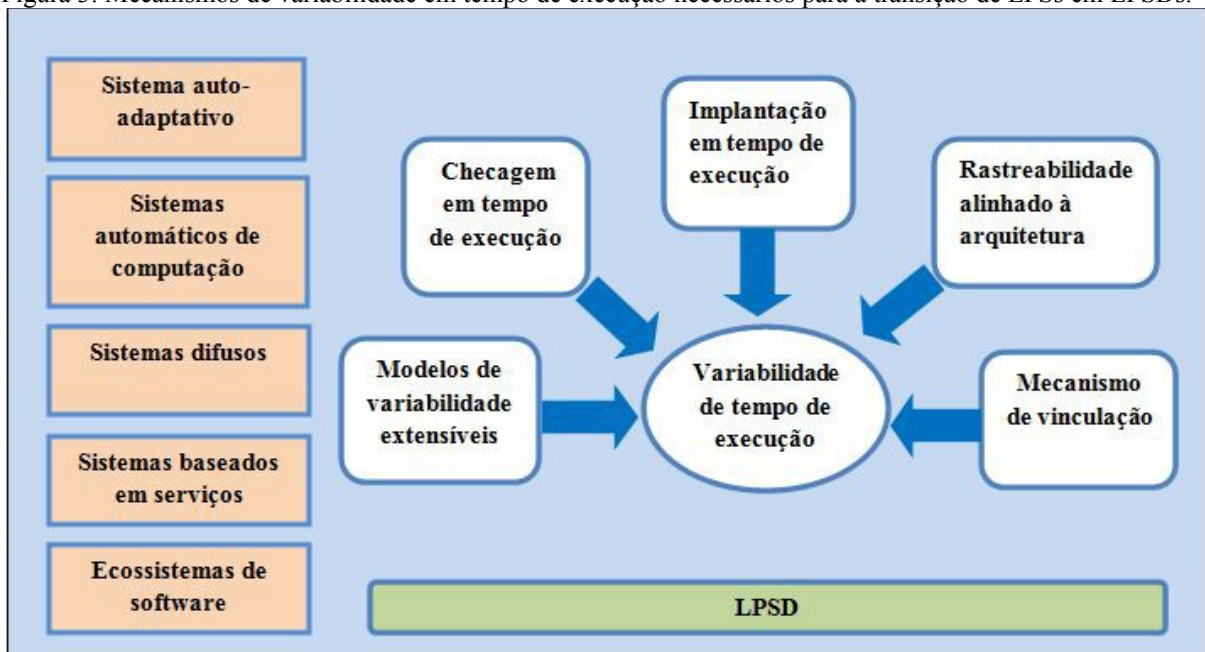
As LPSDs possuem um conceito muito intrínseco à sua essência, a variabilidade em tempo de execução. A variabilidade em tempo de execução consiste de configurações de produto visíveis aos clientes e seus usuários de sistemas, cujo podem escolher uma configuração disponível entre as diversas apresentadas. Diferentemente das LPSDs, o conceito de variabilidade em tempo de projeto, intrínseco às LPSs, caracteriza-se por não expor as configurações ao usuário e já é gerenciada pelo desenvolvedor, na qual se toma decisões conforme os requisitos necessários e gera-se a configuração de produto adequado (BOUZID et al., 2014).

Segundo CAPILLA e BOSCH (2011), há três principais desafios acerca de dar suporte ao desenvolvimento de variabilidade em tempo de execução:

- Representar a variabilidade em tempo de execução requer mudanças de pontos variantes em unidades de software novas e existentes durante o tempo de execução do mesmo, além de possibilitar a reconfiguração automática do sistema;
- Requer uma validação e checagem do novo modelo de *feature* estabelecido automaticamente, buscando a manutenção, consistência e estabilidade do sistema; e
- Uma automática implantação e vinculação de produtos reconfigurados em tempo de execução com um mínimo de interferência.

Segundo a Figura 3 abaixo, demonstra-se os mecanismos importantes para o suporte e transição de LPSs para LPSDs: modelos de variabilidade extensível, e checagem e implantação em tempo de execução, por exemplo.

Figura 3: Mecanismos de variabilidade em tempo de execução necessários para a transição de LPSs em LPSDs.



Fonte: CAPILLA e BOSCH (2011).

Modelos de *features* estáticos não podem modificar características de sistema, ou adaptar-se automaticamente, em tempo de execução. Suas características, as *features*, são criadas e implantadas em tempo de desenvolvimento. Dessa forma, sistemas sensíveis ao

contexto requerem modelos de variabilidade extensível, ou seja, dinâmicos. Pois assim é possível adicionar, remover e modificar *features* em tempo de execução de uma maneira gerenciada (BENCOMO et al., 2012).

Qualquer alteração que possa haver no modelo de *feature* em tempo de execução requer procedimentos de verificação e checagem do modelo. Por exemplo, deve-se adicionar ou remover restrições de modelo a fim de prevenir inconsistências de configuração do produto. Assim, tais mudanças no modelo podem afetar sua estrutura, dependendo talvez de uma reestruturação no modelo de variabilidade (CAPILLA e BOSCH, 2011).

2.3.2 Verificação e Ligação em Tempo de Execução

Em LPSDs podem ocorrer diversas reconfigurações em tempo de execução ou mesmo simples alterações de modelo que requerem uma reestruturação e reimplantação. Contudo, para que possa haver uma verificação e validação dessas configurações de modelo, torna-se necessária a checagem de modelo, que busca garantir que as vinculações, tais como mudanças de adicionar ou remover *features*, não gerem um sistema com configurações inconsistentes (CAPILLA e BOSCH, 2011).

Como atividades essenciais, as LPSDs, além de controlar as adaptações dos produtos, podem monitorar a situação atual do ambiente e dos recursos computacionais (MARINHO, 2012). Assim, como conclusão, segue algumas características das LPSDs, segundo Capilla et al. (2014):

- Suporte e gerenciamento de variabilidade em tempo de execução: deve suportar a ativação e desativação de *features* conforme o gerenciamento de mudanças necessárias; e
- Vinculação dinâmica e múltipla: conforme as mudanças e adaptações necessárias em detrimento das alterações de contexto (propriedades do ambiente), *features* podem ser vinculadas diversas vezes em diferentes momentos.

Em sua essência, as LPSDs são capazes de identificar automaticamente as mudanças de contexto no ambiente em que está executando, realizando adaptações adequadas sem que necessariamente tenham sido definidas durante o seu tempo de desenvolvimento.

2.4 Modelos de *Features*

O modelo de *features*, como mencionado anteriormente, é um artefato de grande importância em uma LPS. Representa as informações, em termos de características e relacionamentos, e onde estão presentes similaridades e variabilidades de produtos de um determinado domínio a serem derivados a partir da LPS. A *feature* pode representar algo diferente dependendo da LPS onde está inserida. Podendo ser uma funcionalidade ou um atributo de qualidade (BEUCHE e DELGARNO, 2007).

O modelo de *features* representa uma família de produtos dentro de uma LPS. Essa representação é feita através de um conjunto de *features* organizados hierarquicamente em formato de árvore, onde cada nó representa uma *feature*. No modelo são também representadas relações entre essas *features*. Para se elaborar o modelo e representar essas relações entre as *features*, existem diferentes notações (BENAVIDES, SEGURA E RUIZ-CORTÉS, 2010). São elas:

- Modelos de *features* básicos

Neste tipo de modelo, são permitidas as seguintes relações entre *features*:

- Obrigatória – Inclusa em todos os produtos derivados da linha;
- Opcional – Pode ou não estar inclusa em um produto;
- Alternativa – Pode ser escolhida dentro de um grupo de *features*, deve ser definido se uma ou mais devem ser escolhidas;
- Or – *Feature* exclusiva, a inclusão de uma *feature* deste tipo em um produto exclui outra.

- Modelos de *features* baseados em cardinalidade

Nesta notação são adicionadas as seguintes relações:

- Cardinalidade de *features*:

Usado para indicar o número de ocorrências da *feature* em um produto através de um intervalo $[n..m]$, onde n é o limite mínimo e m o limite máximo. Por exemplo, $[1,1]$ equivale a dizer que a *feature* é obrigatória. E se a *feature* tem cardinalidade $[0,1]$, equivale a dizer que a mesma é opcional.

- Cardinalidade de grupo de *features*:






















Também existe representação de intervalo $(n..m)$, onde n é o limite mínimo e m é o limite máximo. Esse intervalo limita a quantidade de *features* filhas que devem ser inseridas em um produto caso usa *feature* pai seja selecionada.

- Modelos de *features* estendidos:

Em modelos de *features* estendidos são inseridos informações adicionais sobre cada *feature*. Essas informações são chamadas de atributos, que por sua vez tem a função de especificar informações adicionais além das funcionalidades expressas pelas *features*, como por exemplo, o custo de memória para suportar uma determinada *feature*.

Robak (2003) faz uma comparação entre diferentes métodos e suas notações para a construção de modelos de *features*. Em seu trabalho são mostradas as diferenças entre métodos como FODA – *Feature Driven Domain Analysis*, FORM, GP (*Generative Programming*), Featu-RSEB e J. Bosch. Na Tabela 1 é realizada uma comparação dessas notações.

Tabela 1: Comparação entre notações.

FODA		FORM		GP		Featu-RSEB		J. Bosch		Significado
	Feature Obrigatória		Feature Obrigatória		Obrigatória		Composta de		Composição	Obrigatória (se alcançável, deve ser escolhida).
	Feature Opcional		Feature Opcional		Opcional		Feature Opcional		Opcional	Opcional (se alcançável, pode ser escolhida, ou não).
	Feature Alternativa		Feature Alternativa		Alternativa		Ponto de variação (XOR)		xor - Especialização	Uma-de-muitas Escolha dentre um grupo
-	-	-	-		Features – Or		Ponto de variação (OR)		or – Especialização	n-de-muitas Escolha dentre um grupo
-	-	-	-	-	-	-	-		Feature externa	Feature Externa
-	-	-	-		Feature aberta	-	-	-	-	Feature Aberta
-	-	-	-		Prematura	-	-	-	-	Feature Prematura

Fonte: ROBAK, 2003.

Na notação do método FODA, é possível descrever relações entre *features*, que podem ser Obrigatórias, Opcionais, ou Alternativas. Nessa notação, uma *feature* é obrigatória a menos que haja um círculo vazio em seu nó, nesse caso a *feature* é opcional. Para representar *features* alternativas é usado um arco que se divide em mais de uma *feature*. O método FORM, que é uma extensão de FODA utiliza os mesmos nomes, mas utiliza caixas para os nomes das *features*. O método GP utiliza uma notação que pouco difere de FODA. Nesta notação existem as *features* OR que diferem das alternativas por indicar a possibilidade de escolha de N de muitas *features* dentro do grupo. A notação também possui *features* abertas e prematuras. As abertas são indicadas pela adição de colchetes no nome da *feature*. Esse tipo de *feature* consiste de um grupo XOR que pode ser estendido. As prematuras indicam *features* que ainda requerer maior detalhamento, cuja modelagem ainda não foi finalizada, e

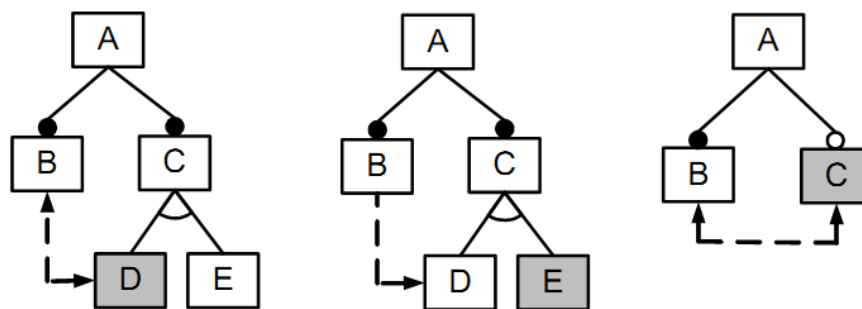
ainda precisa ser mais bem especificada no decorrer da modelagem. A notação usada no método FeatuRSEB é baseada em UML. Traz como diferencial pontos de variação que podem ser do tipo XOR ou OR. Indica um ponto onde variações podem acontecer no design do produto. A notação utilizada por J. Bosch difere das outras por utilizar um triângulo preenchido ou não, para representar *features* alternativas OR ou XOR, respectivamente. Também introduz a noção de especialização, para *features* alternativas (ROBAK, 2003).

No modelo de *features* também podem ser inseridas restrições entre as mesmas. Essas restrições são chamadas *cross-tree*. Podem indicar inclusão ou exclusão de *feature*. A relação de Requires ou Excludes pode ser representada por uma seta com uma linha pontilhada, unidirecional ou bidirecional, respectivamente. Na Figura 4, pode-se ver que no primeiro modelo há uma relação de excludes entre as *features* B e D, indicando que a escolha de uma das *features* implica na exclusão da outra. No segundo modelo, verifica-se entre as *features* B e D a relação de requires, indicando que a inclusão de B em um produto implica na inclusão também de D (BENAVIDES et al., 2010).

Features podem também ter as seguintes classificações:

- *Feature* morta – não está presente em nenhum produto da linha. Seu aparecimento se deve ao um mau uso de restrições *cross-tree* (BENAVIDES et al., 2010). Na Figura 4 é apresentado um exemplo de como uma *feature* morta pode se apresentar no modelo:

Figura 4: Exemplos de *features* mortas.



Fonte: BENAVIDES et al., 2010

As *features* em cinza são consideradas mortas, por causa das restrições inseridas. No primeiro modelo, vê-se que a *feature* B tem uma relação de restrição de exclusão em relação à *feature* D. Como B é obrigatória, B não estará presente em nenhum produto da linha. No segundo modelo, temos que B tem uma relação de restrição com D. Nesse caso, B requer D,

como D e E são alternativas, E não estará presente em nenhum produto da linha. Por fim, o último modelo contém uma relação de restrição de exclusão entre B e C, como B é obrigatória e C opcional, C nunca estará presente em um produto da linha;

- *Feature* comum – está presente em todos os produtos da linha. São elas as *features* obrigatórias, que devem ser inseridas a cada novo produto a ser derivado da LPS (MENDONÇA et al., 2009);
- *Feature top* – *feature* que possui filho. *Features* top possuem *sub-features* na hierarquia do modelo (BEZERRA et al., 2013). Na Figura 4 mostrada anteriormente pode-se considerar como *features* top A e C nos dois primeiros modelos e apenas A no terceiro; e
- *Feature leaf* – *feature* que não possui filho, ou seja, que não possui *sub-feature* na hierarquia do modelo (BEZERRA et al., 2013). Ainda na Figura 4 mostrada anteriormente, pode-se considerar como *features leaf* B, D e E nos dois primeiros modelos e E e C no terceiro.

2.5 Qualidade em LPSs e LPSDs

Em LPS, o princípio base é a reutilização. Assim, é necessária a garantia da qualidade para avaliar os atributos de qualidade referentes às linhas de produto de software, tais como consistência, variabilidade e modularidade, por exemplo, para que possíveis erros sejam identificados e tratados em níveis primários. No mais, a qualidade em linhas de produtos de software com sensibilidade ao contexto tem crescido consideravelmente. Contudo, torna-se um desafio maior garantir a qualidade em um ambiente e sistema com diversas mudanças de requisitos e restrições de qualidade em tempo de execução com as linhas de produto de software dinâmicas. Dessa forma, nas próximas seções serão apresentados sobre a qualidade em LPSs e, por conseguinte, a qualidade em LPSDs.

2.5.1 Qualidade em LPSs

A garantia de qualidade em uma LPS se faz necessária por que em um ambiente de reuso, erros em artefatos podem acarretar erros que se propagarão para diversos produtos que serão derivados a partir deles. O foco da qualidade em LPS está na avaliação de atributos de qualidade dos produtos da linha. Esse processo pode representar um desafio pela variabilidade

presente nos produtos derivados, pois um produto pode demandar um atributo de qualidade diferente de outro (ETXE BERRIA et al., 2008).

Atributos de qualidade de uma LPS podem ser de dois tipos: Atributos de qualidade da LPS e atributos relevantes ao domínio. Os atributos de qualidade da LPS são próprios da linha de produtos, como por exemplo, variabilidade, flexibilidade, reusabilidade da LPS. Atributos relevantes do domínio são dependentes do tipo de domínio em questão, como por exemplo, disponibilidade em um sistema que possui grande fluxo de dados (EXTEBERRIA et al., 2008). Através da análise do modelo de *features* é possível avaliar os seguintes atributos de qualidade de uma de uma LPS: modularidade, consistência, reusabilidade e corretude. Esses atributos estão relacionados à manutenibilidade, característica de qualidade do modelo de *features* (BEZERRA et al., 2013).

Segundo pesquisa a abordagem usada para a avaliação de qualidade em LPS 49% dos casos são técnicas que incluem medidas. Esse mesmo estudo indica que 45% dos métodos utilizados fazem avaliações nos requisitos e fases de design da engenharia de domínio, apenas 26% desses métodos se aplicam à engenharia de aplicação. Vemos daí um indicativo de que avaliações em fases iniciais já é uma tendência na engenharia de LPS (MONTAGAUD e ABRAHAO, 2009).

Tendo isso em vista uma abordagem a se tomar para viabilizar a garantia de qualidade em uma LPS é aplicar medidas de qualidade no modelo de *features*. Sendo esse um artefato essencial no processo de engenharia de domínio, e corrigir erros nesse modelo pode prevenir que erros se propaguem para produtos da linha desenvolvidos posteriormente (BEZERRA et al., 2013).

2.5.2 Qualidade em LPSDs

Com o desenvolvimento de sistemas cada vez mais adaptativos e inteligentes, a dependência em sensibilidade ao contexto aumenta conforme as mudanças de condições no ambiente. Contudo, as LPSs não possuem características adequadas para ser desenvolvido em ambientes que exigem sistemas sensíveis ao contexto, o que também acarreta que a sua garantia de qualidade não abordará aspectos essenciais para esses ambientes. E como extensão das LPS, a garantia da qualidade em LPSDs procura atingir características de contextos, em como as *features* de contexto são analisadas e gerenciadas.

Features de contexto devem ser identificadas, modeladas e gerenciadas para sistemas sensíveis ao contexto, pois o mesmo necessita estar apto para as mudanças de contexto e planos de reconfiguração (FERNANDES et al., 2008). Igualmente, dado que um contexto determina o ambiente em que o sistema está inserido, pode-se usar a variabilidade de contexto para modelar as *features* que serão ativadas ou desativadas em tempo de execução, ou mesmo quando novas funcionalidades são necessárias a serem adicionadas.

Segundo CAPILLA et al. (2014), há duas abordagens para a modelagem de *features* de contexto:

- Dois modelos de *feature* relacionados: a idéia básica dessa estratégia é desenvolver um modelo de *feature* para *features* de contexto e outro modelo para as que não são de contexto. No entanto, essa abordagem pode fazer com que as dependências entre *features* de contexto e não-contexto sobrecarreguem as suas relações. Por exemplo, pode ser necessário que, dada uma *feature* “F”, para cada contexto definido seja necessária a sua definição conforme o valor que ela representa nesse contexto;
- Modelo de *feature* único: nessa estratégia é desenvolvido um único modelo que representa tanto *features* de contexto como não-contexto. Desse modo, é possível reduzir o número de dependências entre as *features*, conforme problemática citada na estratégia anterior. Nesse modelo, as *features* de contexto são rotuladas como *feature* de contexto e classificadas de acordo com um conjunto de tipos predefinidos. Assim, é possível que a autoadaptação aconteça simplesmente por ativar ou desativar *features* de contexto, conforme características de contexto observadas no ambiente.

Sobretudo, como tarefa primordial na garantia de qualidade em LPSDs, a análise de contexto é uma atividade que busca identificar e modelar propriedades de contexto e suas possíveis mudanças em um ambiente de execução. Conforme CAPILLA et al. (2014), da perspectiva de LPSDs, a análise de contexto contém as seguintes atividades: (i) identificar propriedades físicas do sistema: o projetista identifica propriedades pertinentes às mudanças de ambiente ou com o que o sistema interage; (ii) modelagem de *features* de contexto: as propriedades físicas identificadas são modeladas conforme seus contextos seguindo uma das modelagens de *features* anteriormente apresentadas nesse tópico; (iii) identificar *features* que modificam a estrutura do modelo dinamicamente: além das que podem ser ativadas ou

desativadas, deve-se identificar as de contexto que impactam na estrutura do modelo quando há mudanças no sistema em tempo de execução; (iv) definir regras operacionais: além das regras de “requer” e “exclusão” encontradas nas LPSs, é importante existir regras de contexto que direcione a ativação e desativação de *features* de contexto em tempo de execução.

Contudo, verificamos que a garantia de qualidade nas LPSDs torna-se mais complexa pela existência de características presentes em sistemas sensíveis ao contexto, pois tais sistemas estão vinculados às ações e mudanças em tempo de execução, exigindo que os modelos de *features* possuam aspectos que se adequem e reestruturem conforme as exigências do ambiente, requerendo novos processos de análise e reconfiguração do modelo para tais ajustes.

2.5.3 Medidas de Qualidade do Modelo de *Features*

Segundo Bezerra et al. (2014), “uma medida de qualidade é um mapeamento de uma entidade para a representar em número ou símbolo, de forma a caracterizar uma determinada propriedade do mesmo”. Dessa forma, as medidas de qualidade tornam-se importantes para prover resultados a respeito de atributos inerentes ou específicos de linhas de produtos de software, tais como a flexibilidade ou a variabilidade.

Em revisão sistemática, Montagud et al. (2009), identificaram 165 medidas relacionadas com 97 atributos de qualidade diferentes. Mais de 90% dessas medidas avaliam atributos relacionados à manutenibilidade. O estudo também indica que 67% dessas medidas são usadas durante a engenharia de domínio.

Em Bezerra et al. 2014, é feita uma revisão bibliográfica onde foram identificadas diversas medidas para a avaliação da qualidade do modelo de *features*. Os atributos de qualidade da LPS identificados são modularidade, reusabilidade, tamanho do modelo de *features*, validade das configurações, flexibilidade da configuração, profundidade da árvore, complexidade ciclomática, conectividade entre os componentes do modelo de *features*, grau de envolvimento das *features* na definição das restrições de integridade e corretude. Tais atributos de qualidade estão relacionados à manutenibilidade, eficiência de desempenho, adequação funcional, usabilidade, portabilidade, confidencialidade e segurança da LPS. As medidas que atendem a cada uma desses atributos são descritos na Tabela 2.

Tabela 2: Características e atributos de qualidade das medidas identificadas para avaliação da qualidade do modelo de *features*.

Características	Atributo de qualidade	Medida
Adequação funcional	Corretude funcional	Precision / Recall / F-measure
	Adequação funcional	Valor de importância da <i>feature</i> .
Manutenibilidade	Analísabilidade	Número de <i>features</i> folhas. Impacto da mudança.
	Instabilidade	Flexibilidade de mudança. Índice de manutenção de uma <i>feature</i> . Impacto quantitativo de <i>features</i> variáveis em atributos de qualidade.
	Complexidade cognitiva	Complexidade cognitiva de um modelo de <i>feature</i> .
	Extensibilidade	Extensibilidade de <i>Feature</i> .
	Flexibilidade	Flexibilidade de Configuração
	Modularidade	<i>Features</i> dependentes de ciclos únicos. <i>Features</i> dependentes de ciclos múltiplos.
	Reusabilidade	Comunalidade não funcional
	Complexidade estrutural	Complexidade ciclomática. Complexidade de configuração. Complexidade de restrição. Complexidade estrutural. Complexidade composta. Complexidade de variabilidade. Complexidade de variante. Restrições. Profundidade da árvore. Número de <i>features</i> . Número de <i>features</i> topo.
	Variabilidade	<i>Features</i> de pontos de variação múltiplos. Número de <i>features</i> variáveis. Número de pontos variantes. Taxa de variabilidade. Número de configurações válidas. <i>Features</i> de pontos de variação rígidos. <i>Features</i> de pontos de variação únicos.
	Usabilidade	Facilidade de Uso
Eficiência de desempenho	Precisão	-
	Utilização de recurso	-
	Escalabilidade	-
	Comportamento de tempo	Tempo de documentação. Tempo quando uma <i>feature</i> foi incluída no escopo do projeto.
Portabilidade	Adaptabilidade	Adaptabilidade estática de <i>feature</i> . Adaptabilidade dinâmica de <i>feature</i>
	Disponibilidade	-

Confiabilidade	Consistência	Taxa de consistência
Segurança	Autenticidade	-
	Integridade	-

Fonte: BEZERRA et al, 2014.

Para cada medida é definida uma fórmula pra a obtenção dos resultados da aplicação da medida nos modelos de *features*. Na Tabela 3 são descritas com mais detalhes as medidas e são apresentadas suas respectivas fórmulas.

Tabela 3: Conjunto de medidas identificadas para avaliar a qualidade do modelo de *features*.

Medida	Descrição	Fórmula de Cálculo
Número de <i>features</i> (<i>NF</i>)	O número total de <i>features</i> presentes no modelo.	$NF = \sum (\text{Número de } features \text{ do modelo de } features)$
Número de <i>features</i> folha (<i>NLeaf</i>)	O número de <i>features</i> com nenhuma <i>feature</i> filha ou especialização.	$NLeaf = \sum (\text{Número de } features \text{ sem filhos do modelo de } features)$
Complexidade cognitiva (<i>CogC</i>)	Denota quão fácil um <i>software</i> pode ser compreendido, que relata a variabilidade em Engenharia de Linhas de produto.	$CogC = \sum (\text{Número de pontos variantes})$
Flexibilidade de Configuração (<i>FoC</i>)	Esta é a razão entre o número de funcionalidades opcionais ao longo de todas as <i>features</i> disponíveis no modelo de <i>feature</i> . A lógica por trás disso é indicar a possibilidade de gerar configurações distintas.	$FC = NFO/NF$ NFO - Número de <i>Features</i> Opcionais NF - Número de <i>Features</i>
<i>Features</i> dependentes de ciclos únicos (<i>SCDF</i>)	Número de <i>features</i> que estão presentes nas restrições do modelo e filhas de pontos de variação com cardinalidade [1..1].	$SCDF = \sum (\text{Número de } features \text{ participantes em Constraints e filhas de pontos variantes com cardinalidade [1..1]})$
<i>Features</i> dependentes de ciclos múltiplos (<i>MCDF</i>)	Número de <i>features</i> que estão presentes nas restrições do modelo e filhas de pontos de variação com cardinalidade [1..*].	$MCDF = \sum (\text{Número de } features \text{ participantes em Constraintse filhas de pontos variantes com cardinalidade [1..*]})$
Extensibilidade de <i>Feature</i> (<i>FEX</i>)	Refere-se ao número de <i>features</i> que podem ser facilmente adicionadas ao modelo durante a fase de manutenção.	$FEX = MCDF + SCDF + NLeaf$
Complexidade ciclomatica (<i>CyC</i>)	O número de ciclos diferentes que podem ser encontrados no modelo de <i>features</i> . Uma vez que os modelos de <i>features</i> são sob a forma de árvores, sem ciclos podem existir em um modelo de <i>features</i> , no entanto, as restrições de integridade entre os <i>features</i> disponíveis pode causar ciclos. É simples para mostrar que o número de ciclos distintos de complexidade e, portanto, de um modelo de ciclomática funcionalidade é equivalente ao número de restrições de integridade de um modelo de	$CyC = \sum (\text{número de restrições de integridade de um modelo de } features)$

	<i>features</i> .	
--	-------------------	--

Fonte: BEZERRA et al, 2014.

Tabela 4: Continuação da Tabela 3.

Medida	Descrição	Fórmula de Cálculo
Complexidade composta (ComC)	Medida composta que representa o número de pontos variantes, o número de <i>features</i> variáveis, número de restrições de integridade e seus relacionamentos.	$ComC = F^2 + (Rand^2 + 2Ror^2 + 3Rcase^2 + 3Rgr^2 + 3R^2)/9$ $F = \text{Número de } features$ $Rand = \text{Número de Relações mandatórias}$ $Ror = \text{Número de Relações de agrupamento OR}$ $Rcase = \text{Número de Relações de agrupamento XOR}$ $Rgr = \text{Número de Relações de agrupamentos R}$ $R = \sum (\text{Número de Relações de agrupamento}) + \sum (\text{Número de Constraints})$
Cross-tree Constraints (CTC)	A razão entre o número de <i>features</i> únicas envolvidas na restrição de integridade do modelo de <i>features</i> sobre o total do número de <i>features</i> do modelo de <i>features</i> . Essa medida representa o grau de envolvimento das <i>features</i> na definição das restrições de integridade.	$CTC = NFRI / NF$ $NFRI = \text{número de características únicas envolvidas na restrição de integridade do modelo de } features$ $NF = \text{Número de Features}$
Profundidade da árvore (DT)	O comprimento do caminho mais longo a partir da raiz do modelo de <i>features</i> até a <i>feature</i> folha do modelo de <i>features</i> .	$DT = \sum (\text{Número de } features \text{ do maior caminho a partir da raiz do modelo de } features \text{ até a } feature \text{ folha do modelo de } features)$
Comunalidade de <i>features</i> não funcionais (NFC)	Representa as <i>features</i> que estão presentes em todas as configurações geradas pelo modelo.	$NFC = \text{Número total de } features \text{ não funcionais comuns (obrigatórias) do modelo de } features / \text{Número total de } features$
<i>Feature</i> de ponto variante múltipla (MHF)	Representa o número de <i>features</i> que podem ser adicionadas em tempo de execução através de pontos de variação com cardinalidade [1..*]	$MHF = \sum (\text{Número de } features \text{ filhas de pontos variantes com cardinalidade [1..*]})$
Número de configurações válidas (NVC)	O número de todas as configurações possíveis e válidas que podem ser derivadas a partir do modelo de <i>feature</i> em face de restrições a sua integridade e estrutura de árvore.	$NVC = \sum (\text{Número de configurações possíveis e válidas do modelo de } feature)$
Número de <i>features</i> variáveis (NVF)	O número de <i>features</i> presente no modelo que estão relacionadas através de <i>features</i> de ponto variante.	$NVF = A + B/N$ $A = \text{Número de } features \text{ alternativas}$ $B = \text{Número de } features \text{ opcionais}$ $N = \text{Número de } features \text{ totais}$
Taxa de variabilidade (RoV)	O fator médio de ramificação que a <i>feature</i> pai apresenta no modelo de <i>features</i> . Em outras palavras, o número médio de filhos dos nós na árvore do modelo de <i>features</i> .	$RoV = \sum (\text{Número médio de filhos dos nós no modelo de } features)$
<i>Features</i> não ponto variantes rígidas (RnHf)	O Número de <i>features</i> que podem ser adicionadas em tempo de execução, mas que não seja através de <i>features</i> de ponto variante.	$RnHF = \sum (\text{Número de } features \text{ não filhas de pontos variantes})$
<i>Features</i> de ponto variante única (SHF)	Representa o número de <i>features</i> que podem ser adicionadas em tempo de execução através de pontos de variação com cardinalidade [1..1]	$SHF = \sum (\text{Número de } features \text{ filhas de pontos variantes com cardinalidade [1..1]})$

Fonte: BEZERRA et al, 2014.

2.6 Ferramentas para Avaliação da Qualidade do Modelo de *Features*

Para o desenvolvimento desse trabalho proposto, uma análise exploratória foi feita em busca de ferramentas de análise em LPS que já foram desenvolvidas e que pudessem servir de base e orientar. Dessa forma, encontrou-se ferramentas que manipulavam modelos de *features* e que implementam medidas de qualidade para avaliação das mesmas. Tais ferramentas serão descritas nas próximas seções, o S.P.L.O.T na primeira seção e o FAMILIAR em sequência.

2.6.1 Ferramenta S.P.L.O.T.

A ferramenta S.P.L.O.T.¹, utilizada para criação e edição de modelos de *features* online, é baseada em web, desenvolvida em Java (JAVA, 2014) com HTML (HTML, 2014). Disponibiliza um banco de dados de modelos de *features* que pode ser utilizado por pesquisadores da área de LPS (MUNIR; SAHID, 2010). Na ferramenta temos disponíveis as seguintes funcionalidades:

- i. **Editor de Modelo de *features*:** nesse editor, ilustrado na Figura 5, é possível criar um novo modelo de *features* ou editar um existente. Para a criação existem instruções que guiam o usuário nessa tarefa. Além disso, os modelos criados podem ser salvos no repositório da ferramenta;

¹ <http://www.splot-research.org/>

Figura 5: Editor de modelos de *features*

⚠ All information will be lost if you exit this page. Hence, make sure you export or save your model regularly.

Feature Diagram

Type name for root feature

Cross-Tree Constraints

Click to create a constraint

Additional Information

Name your feature model: (*)

Short description of feature model: (*)

Primary Author: (*)

Author's Address:

Author's Email: (*)

Author's Phone Number:

Author's Website:

Author's Organization: (*)

Author's Organization Department:

Date model was created:

Where was model published? (if applicable):

(*) Mandatory fields if you wish to add your model to SPLOT's feature model repository

Feature Information Table

Id:

Name:

Description:

Type:

#Children:

Tree level:

Update Feature Model

Feature Model Statistics

#Features	1
#Mandatory	0
#Optional	0
#XOR groups	0
#OR groups	0
#Grouped	0
#Cross-Tree Constraints (CTC)	0
CTCR (%)	0.00
#CTC distinct vars	0
CTC clause density	0.00

Feature Model Analysis

Consistency

Dead Features

Core Features

Valid Configurations

Run Analysis

Run Analysis every time I ask for

Fonte: S.P.L.O.T. 2013

- ii. **Análise automatizada:** Na página de análise automatizada, ilustrada na Figura 6, pode-se fazer a análise de um modelo salvo na ferramenta ou o seu próprio modelo criado em SXFM (*Simple XML Feature Model Fomat*), que não tenha sido salvo no repositório. Apenas modelos consistentes com 10 *features* ou mais e sem *features* mortas são salvos no repositório. Na análise automatizada existem diversas medidas relevantes para a avaliação da qualidade do modelo. São elas:
- Número de *features*, opcionais, obrigatórias e agrupadas;
 - Profundidade da árvore;
 - Consistência do modelo;
 - *Features* comuns;
 - *Features* mortas;
 - Número de configurações válidas;
 - Grau de variabilidade;
 - Complexidade ciclomática;
 - Cross-tree constraints;
 - Densidade CTC.

Figura 6: Análise automatizada

Data	Telecommunication_System (view)
Statistics	
#Features	12
- Optional	3
- Mandatory	3
- Grouped	5
- Groups	2
Tree Depth	4
ECR (%)	33
#Extra constraints	2
#Distinct extra constraints variables	4
Clause Density	0.5
#CNF Clauses	23
Debugging Analyses (Click to Run the SAT solver)	
Consistency	consistent
Running Time (ms)	0
#Dead Features	0
- Running Time (ms)	0
#Common Features	3 view
- Running Time (ms)	1

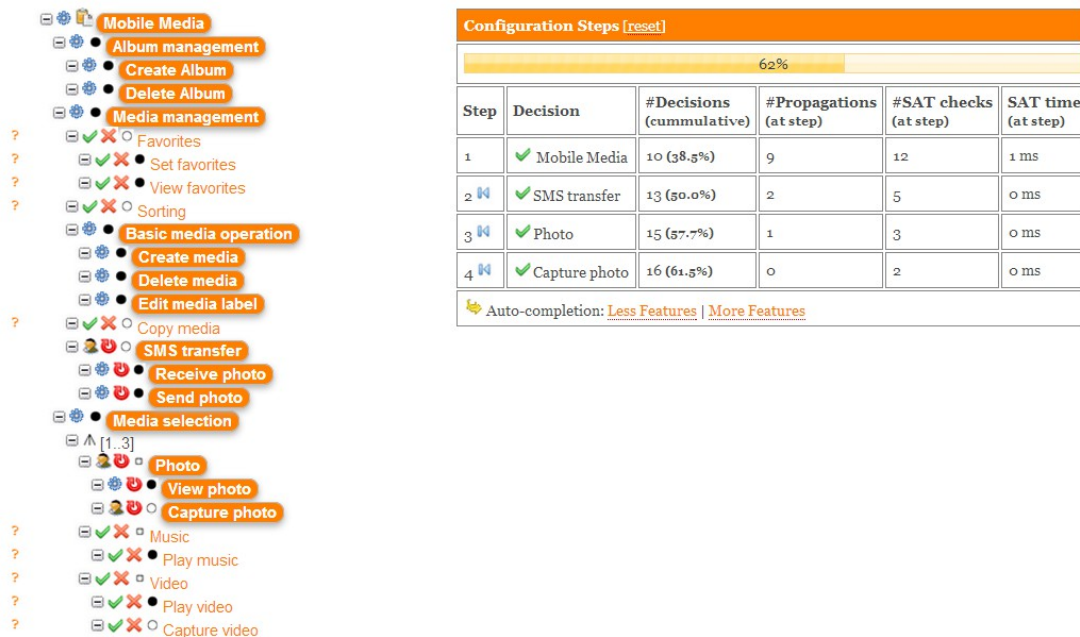
Fonte: S.P.L.O.T., 2013

Ao fazer a análise de um modelo, essas medidas são apresentadas em uma tabela, conforme a Figura 6, mostrando todos os valores gerados pela análise. O objetivo desse trabalho é adicionar as novas medidas e apresentar nessa ferramenta com os valores fornecidos pelas medidas.

- iii. **Configuração de produto:** como apresentado na Figura 7, é possível configurar um novo produto, a partir de um modelo de *features* do repositório ou um modelo do próprio usuário. A configuração consiste em adicionar *features* do modelo, criando assim um produto único;

Figura 7: Configuração de produto

Mobile Media v8 (26 features)



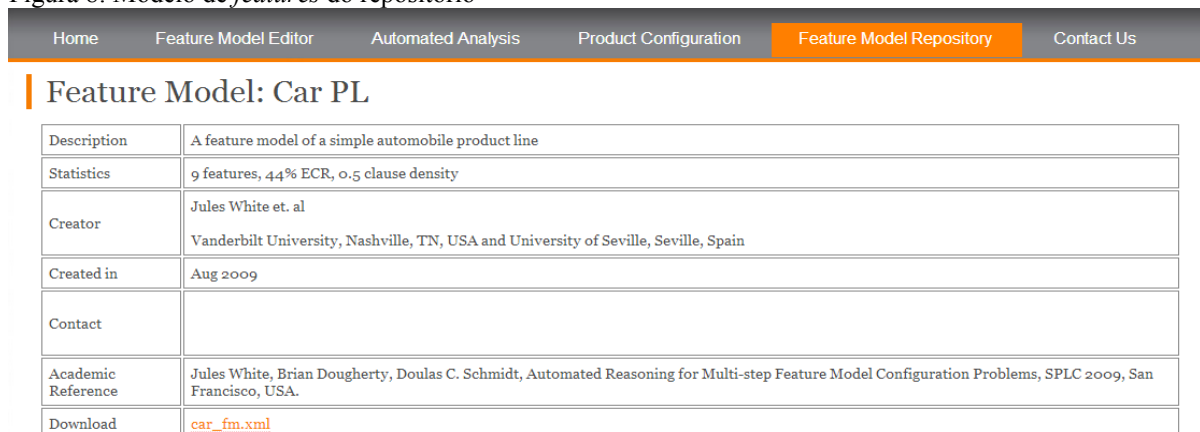
The screenshot shows the configuration interface for 'Mobile Media v8'. On the left, a tree view lists 26 features, including Album management, Media management, Basic media operation, SMS transfer, and Media selection. On the right, the 'Configuration Steps' table shows the progress of the configuration process, with a 62% completion bar at the top.

Step	Decision	#Decisions (cumulative)	#Propagations (at step)	#SAT checks (at step)	SAT time (at step)
1	Mobile Media	10 (38.5%)	9	12	1 ms
2	SMS transfer	13 (50.0%)	2	5	0 ms
3	Photo	15 (57.7%)	1	3	0 ms
4	Capture photo	16 (61.5%)	0	2	0 ms

Auto-completion: [Less Features](#) | [More Features](#)

Fonte: S.P.L.O.T. 2013

- iv. **Repositório de Modelos de *features*:** O repositório, ilustrado na Figura 8, conta com quase quatrocentos modelos prontos disponíveis para *download*, edição ou configuração de produto. Apenas são armazenados no repositório modelos consistentes, que possuam pelo menos uma configuração válida e modelos corretos, que não possuam *features* mortas cujos autores sejam devidamente identificados e forneçam alguma informação de contato.

Figura 8: Modelo de *features* do repositório


The screenshot shows the 'Feature Model Repository' interface. The top navigation bar includes 'Home', 'Feature Model Editor', 'Automated Analysis', 'Product Configuration', 'Feature Model Repository', and 'Contact Us'. The main content area displays the details for a 'Feature Model: Car PL'.

Description	A feature model of a simple automobile product line
Statistics	9 features, 44% ECR, 0.5 clause density
Creator	Jules White et. al Vanderbilt University, Nashville, TN, USA and University of Seville, Seville, Spain
Created in	Aug 2009
Contact	
Academic Reference	Jules White, Brian Dougherty, Douglas C. Schmidt, Automated Reasoning for Multi-step Feature Model Configuration Problems, SPLC 2009, San Francisco, USA.
Download	car_fm.xml

[back](#)

Fonte: S.P.L.O.T. 2013

Na página da ferramenta também é possível encontrar diversas notícias sobre as pesquisas na área. Também é possível baixar o código da ferramenta para modificar (S.P.L.O.T., 2013).

2.6.2 Familiar

A segunda ferramenta a ser apresentada nesse trabalho é a FAMILIAR², uma ferramenta desenvolvida em JAVA para criação e edição de modelos de *features*. Na verdade, FAMILIAR (FAMILIAR, 2014) caracteriza-se por ser uma linguagem de *script* (CELES et al., 2004) criada para manipulação e análise automática de modelos de *feature*, complementando o software com interface gráfica disponível ao usuário.

Seguindo o mesmo processo de apresentação feito com a ferramenta anterior, temos as seguintes funcionalidades:

- i. **Criação e Edição de Modelo de *features*:** FAMILIAR é uma linguagem com tipos de dados predefinidos, logo ele define o que será um Modelo de *Feature* ou uma *Feature*, por exemplo. Na Figura 9 abaixo mostra a operação de criação de um modelo *feature*.

Figura 9: Operações de criação e análise.

```

1 gc1 = FH ( GraphicCard: DirectX Speed MemoryBus [Multi];
2             DirectX: (v10dot1 | v10) ; Speed: (n800 | n1000) ; Bus: n128 ;
3 gc2 = FH ( GraphicCard: DirectX Speed MemoryBus [Multi];
4             DirectX: (v10dot1 | v10) ; Speed: (n800 | n1000) ; Bus: n128 ;
5 b1 = gc1 eq gc2 // b1 is true
6 b2 = gc1 == gc2 // b1 is false
7 str = "v10" // "==" and "eq" are equivalent for strings
8 fmSet = {gc1, gc2}

```

Fonte: FAMILIAR (2014).

Como podemos observar, as duas variáveis “gc1” e “gc2”, são criadas como um tipo modelo de *feature* possuindo a estrutura inserida conforme a imagem. Logo abaixo da criação, são utilizadas operações simples da linguagem FAMILIAR que verificam a igualdade dos modelos, por exemplo.

- ii. **Análise de estrutura do modelo de *feature*:** com a linguagem de *script* é possível fazer análises sobre a estrutura do modelo de *feature*. Segue a Figura 10 apresentando algumas operações.

² <https://nyx.unice.fr/projects/familiar>

Figura 10: Operações de análise estrutural do modelo.

```

1 gc3 = FM ( GraphicCard: DirectX Speed Bus Multi ;
2           DirectX: (v11 | v10dot1) ; Speed: n1000 ; Bus: n256 ; )
3 f1 = parent v11 // f1 refers to feature named 'DirectX' in gc3
4 f2 = root gc3 // f2 refers to feature named 'GraphicCard' in gc3
5 s1 = name f2 // s1 is a string "GraphicCard"
6 fs = children f1 // fs refers to the set of features named 'v11' and 'v10dot1' in gc3

```

Fonte: FAMILIAR (2014).

De acordo com a imagem, podemos visualizar operações que verificam a estrutura do modelo de *feature* buscando, por exemplo, quem é o pai de uma determinada *feature*, quem é a raiz do modelo e quem são os filhos de uma *feature* específica.

- iii. Configuração de produto:** é possível também criar uma configuração específica de produto do modelo de *feature* criado.

Figura 11: Operações de configuração de produto.

```

1 conf1 = configuration gc1 // create a configuration of gc1
2 b1 = select Multi in conf1 // feature Multi of gc1 is selected
3 b2 = deselect Multi in conf1 // override the previous selection
4 b3 = unselect Multi in conf1 // Multi is neither selected nor deselected

```

Fonte: FAMILIAR (2014).

Na Figura 11, podemos perceber operações que crie uma determinada configuração de produto do modelo de *feature*. Além disso, é possível editar a atual configuração, selecionando ou retirando uma determinada *feature* da configuração gerada.

- iv. Análise automatizada:** com a linguagem FAMILIAR pode-se fazer análises de qualidade de um determinado modelo de *feature*, utilizando algumas medidas já definidas pela própria ferramenta. São medidas encontradas na ferramenta:

- Verificar se uma configuração específica de produto é válida;
- Quantidade de configurações válidas;
- Quantidade de *features* no modelo;
- Profundidade da árvore do modelo; e
- *Features* mortas.

E na Figura 12 abaixo, apresentam-se algumas das operações de análise automatizada disponíveis na linguagem:

Figura 12: Operações de análise automatizada.

```

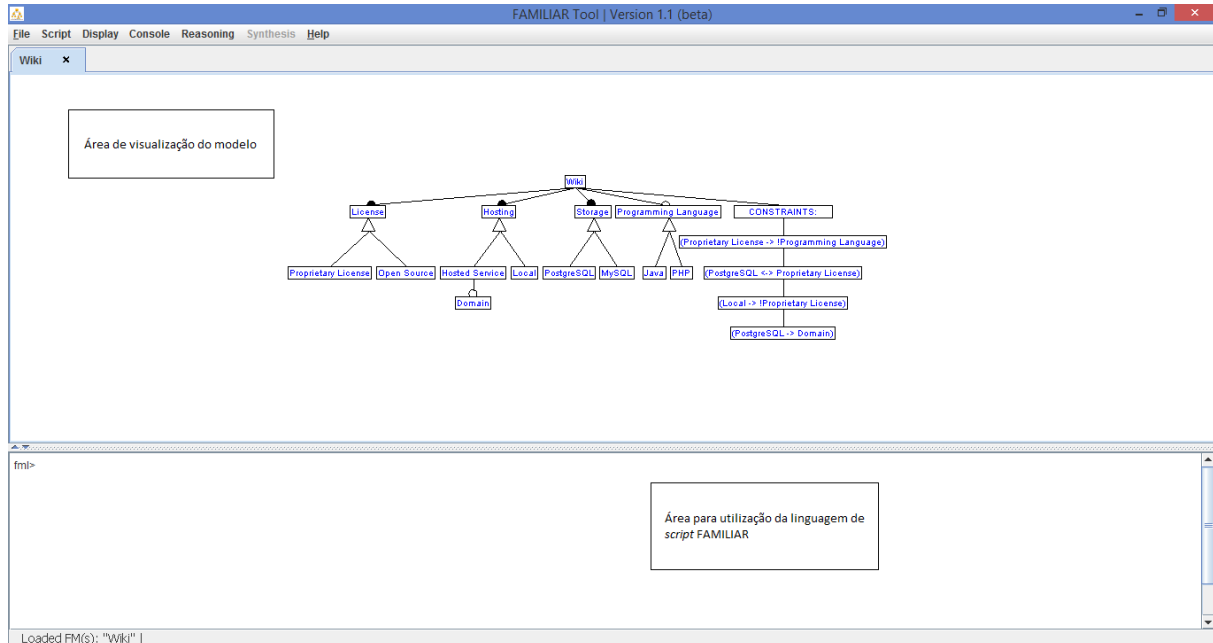
1 conf1 = configuration gc1 // create a configuration of gc1
2 b1 = select Multi in conf1 // feature Multi of gc1 is selected
3 b2 = deselect Multi in conf1 // override the previous selection
4 b3 = unselect Multi in conf1 // Multi is neither selected nor deselected

```

Fonte: FAMILIAR (2014).

Além das funcionalidades apresentadas, como ferramenta, há também uma interface gráfica disponível para os usuários, provendo maiores facilidades de visualização e utilização. Segue a Figura 13 que apresenta de forma geral a ferramenta.

Figura 13: Visão geral da interface gráfica do FAMILIAR.



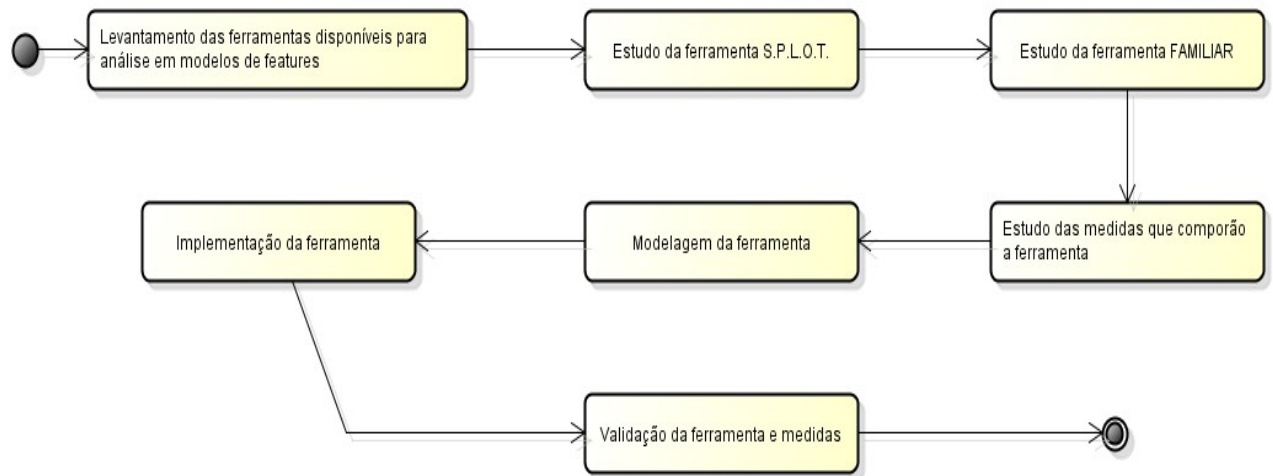
Fonte: elaborado pelo autor.

Constatamos na Figura 13 duas áreas: área de visualização de modelo e área para utilização da linguagem de *script*. E, além das duas áreas principais, podemos perceber um menu que oferece algumas funcionalidades, como: (i) *file*: criar novos modelos, bem como importar ou exportar modelos; (ii) *script*: executar um *script* preconstituído; (iii) *reasoning*: opção onde encontramos as medidas disponíveis pela ferramenta.

3 PROCEDIMENTOS METODOLÓGICOS

Esta Seção apresenta os procedimentos para a completude do trabalho. Como ilustrado na Figura 14, primeiramente será feito um levantamento de ferramentas para análise em modelos de *feature*, e, em seguida, o estudo das mesmas, tanto do código quanto do funcionamento da ferramenta em si. Em seguida, as medidas propostas serão estudadas e então projetar os algoritmos para implementá-las, baseando-se nas fórmulas especificadas na Seção 2.5.3. Após isso, as medidas serão implementadas em código na própria ferramenta. Por fim, a aplicação será modelada e desenvolvida, constando também de uma etapa de validação.

Figura 14: Procedimentos que serão realizados durante a execução deste trabalho



Fonte: Elaborado pelo autor.

3.1 Levantamento das Ferramentas Disponíveis para análise em modelos de *features*

Nesta etapa do projeto, foram identificadas ferramentas já desenvolvidas para análise de modelos de *features*. Neste processo também foi verificado quais medidas de qualidade em modelos de *features* eram contempladas pelas ferramentas. No final, foram identificadas as ferramentas S.P.L.O.T (S.P.L.O.T, 2014) e FAMILIAR (FAMILIAR, 2014).

3.2 Estudo da ferramenta S.P.L.O.T

Para a criação da ferramenta proposta neste trabalho, tal ferramenta foi estudada de forma a criar uma base de conhecimento em seu funcionamento e suas medidas contempladas na análise de qualidade do modelo de *feature*, possibilitando que as mesmas e novas medidas sejam abordadas na ferramenta proposta. Com esse objetivo, foi necessário analisar o código-fonte da ferramenta em busca das medidas, conforme citadas no texto anteriormente, e como são implementadas.

A partir da identificação dessas medidas no código, o processo de codificação das medidas não contempladas pela ferramenta fica mais fácil, pois se poderá identificar que outros recursos do código estão sendo usados para o desenvolvimento dessas medidas já implementadas e quais poderão ser utilizados nestas novas a serem codificadas.

Em tal estudo, verificou-se que o S.P.L.O.T utiliza-se de uma biblioteca para análise automatizada dos modelos, a S.P.L.A.R (*Software Product Lines Automated Reasoning*

library) (MENDONÇA, 2009), que provê algoritmos baseados em SAT (*Boolean Satisfiability Problem*) (ZHANG et al., 2001) e BDD (*Binary Decision Diagrams*) (FRIEDMAN e SUPOWIT, 1987), implementando heurísticas de ordem variável adaptadas à modelos de *features* buscando a otimização da estrutura em BDD. Bem como, a biblioteca abrange vários algoritmos baseados em SAT para fornecer medidas como a computação de domínios válidos, a detecção de conflitos, entre outros.

Segundo ZHANG et al. (2001), “O Problema de Satisfabilidade Booleana (SAT) é um dos problemas NP-Completo mais estudado devido à sua importância tanto em pesquisa teórica quanto em aplicações práticas”. Dada uma formula booleana, o SAT determina se há alguma interpretação que a satisfaça. Contudo, conforme BRYANT (1986), BDD são estruturas de codificação compactas para fórmulas booleanas que oferecem diversos algoritmos de raciocínio eficientes.

3.3 Estudo da ferramenta FAMILIAR

Após a exploração e amadurecimento com o estudo da ferramenta S.P.L.O.T., a ferramenta FAMILIAR foi analisada seguindo os mesmo princípios. Igualmente à ferramenta anterior, a FAMILIAR possui sua interface para criação e edição dos modelos de *features*, assim, além do estudo utilizando-se da interface de usuário, foi analisada o funcionamento da própria linguagem de *script* da ferramenta. Para complementar, foram verificadas quais medidas de qualidade em modelos de *features* a ferramenta contempla.

3.4 Estudo das medidas que comporão a ferramenta

As medidas para avaliação da qualidade do modelo de *features* são parte de um trabalho de doutorado, e foram publicadas em Bezerra et al. (2013) e Bezerra et al. (2014). As medidas já possuem descrição e fórmula de cálculo das medidas, necessárias para realizar a implementação das mesmas para que se possam definir os algoritmos. Dessa forma é possível que a coleta das medidas seja realizada de forma automática e possa realizar a avaliação a partir dos dados coletados da ferramenta.

3.5 Modelagem da ferramenta

Com base nos estudos das ferramentas anteriormente apresentadas e funcionalidades levantadas, identificaram-se requisitos funcionais importantes para serem aplicados na ferramenta proposta neste trabalho. Assim, alguns requisitos levantados foram com base nas funcionalidades comuns nas aplicações estudadas e, para resultar no trabalho em análise de LPSD, outras funcionalidades de tratamento para os modelos *features* adicionando contexto foram inseridas.

3.6 Implementação da ferramenta

A ferramenta foi desenvolvida na tecnologia JAVA, especificamente na plataforma J2SE, de forma que pudesse atender aos usuários localmente no seu computador, sem a necessidade de internet.

Contudo, foram desenvolvidos algoritmos que pudessem fazer a leitura de arquivos que definiam a especificação de modelos de *features* para a ferramenta S.P.L.O.T., permitindo a visualização de tais modelos na ferramenta proposta. Da mesma forma, desenvolveu-se uma área específica para edição dos modelos, abrangendo então a proposta de inserção de contexto nos modelos de *features* importados.

Após os estudos das medidas que serão contempladas, bem como ter-se projetado os algoritmos para cada medida e desenvolvido a estrutura da ferramenta proposta neste trabalho, foi feita a implementação das novas medidas de qualidade em modelos de *feature* propostas, além da possibilidade de exportar os resultados das medidas de qualidade para serem analisados em uma aplicação externa.

3.7 Validação das medidas

Após a finalização da implementação e teste das novas medidas na ferramenta, elas serão aplicadas em modelos de *features* diversificados, podendo ser encontrados no próprio repositório da ferramenta S.P.L.O.T. A validação será feita através da comparação dos resultados obtidos com as medidas aplicadas aos modelos.

4 PROJETO E DESENVOLVIMENTO DA FERRAMENTA

Nesta seção serão abordadas as atividades de projeto e desenvolvimento da ferramenta de análise e edição de modelos de *features* de LPSD.

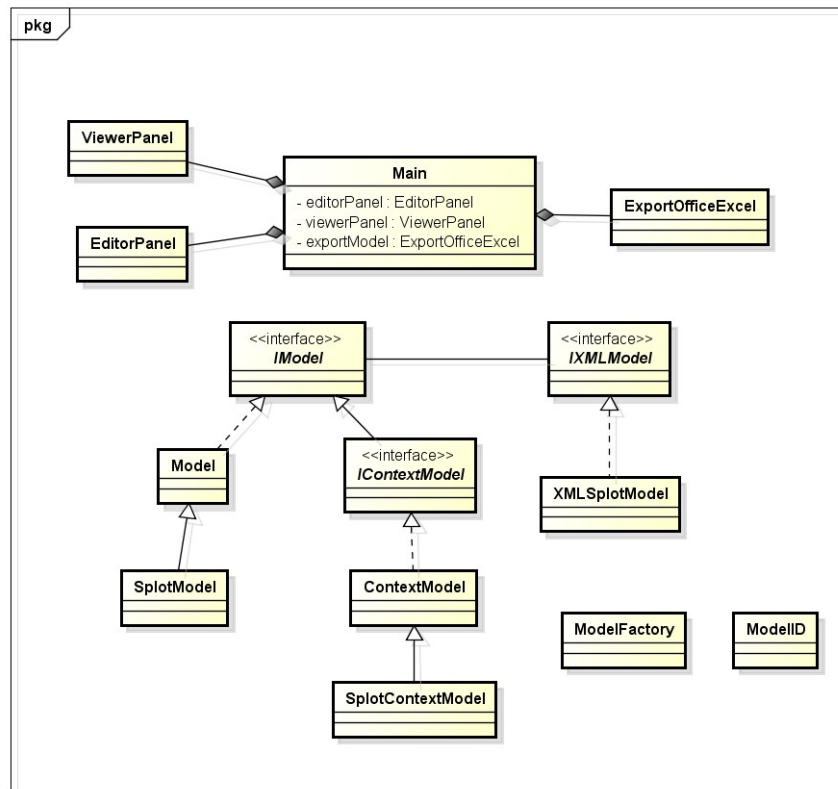
4.1.1 Requisitos funcionais

Primeiramente foram definidos os requisitos funcionais da ferramenta. Estes requisitos foram baseados nas necessidades de avaliação e edição da ferramenta para o modelo de *features* das LPSDs. Segue as funcionalidades da aplicação:

- Importação de modelo de *feature* com base em estrutura de arquivos gerados pelo S.P.L.O.T.;
- Visualização do modelo de *feature* conforme o contexto selecionado, também exibido pela aplicação;
- Aplicação de diferentes medidas de qualidade sobre o modelo de *feature* e seus contextos;
- Área de edição do modelo, onde é possível criar novos contextos e definir *features* ativas ou desativadas; e
- Exportar resultados das medidas de qualidade, baseados em diversos modelos de *features* sem contexto ou resultado das medidas de um modelo e seus contextos.

Também foi definido o diagrama de classes da ferramenta, conforme a Figura 15. O Diagrama de Classes representa os componentes da ferramenta, guia o desenvolvimento dos requisitos funcionais da aplicação conforme a estrutura apresentada.

Figura 15: Imagem do diagrama de classes.

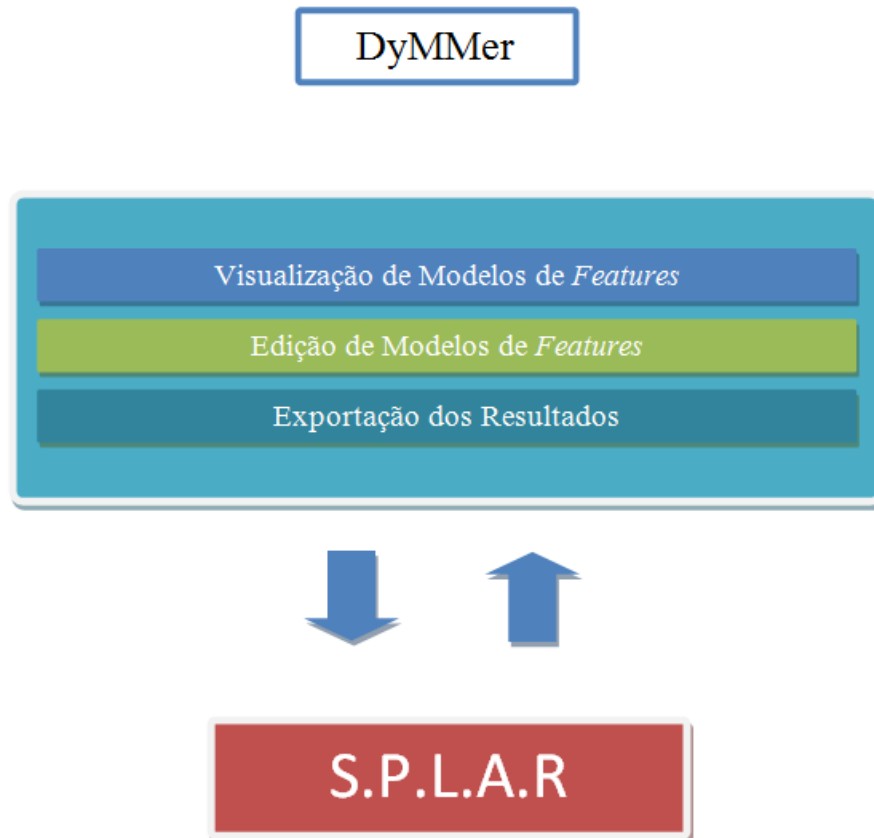


Fonte: elaborada pelo autor.

4.1.2 Arquitetura

Para dispor das funcionalidades recém citadas, uma arquitetura foi definida de forma a relacionar componentes necessários à completude da ferramenta. Como base, a tecnologia JAVA SE (JAVA, 2014) foi a plataforma utilizada para o desenvolvimento da aplicação gráfica, permitindo desenvolver as funções e componentes necessários à aplicação. Dessa forma, a ferramenta aborda mais três camadas: (i) a camada para exportação dos dados: permite a exportação dos resultados das medidas aplicadas aos modelos e seus contextos; (ii) camada de visualização de modelos: possibilita a visualização do modelo de *feature*, bem como os contextos compreendidos; e (iii) a camada de edição dos modelos: habilita criar novos contextos e regras de restrições conforme edita-se o modelo. A Figura 16 mostra como se estrutura a ferramenta.

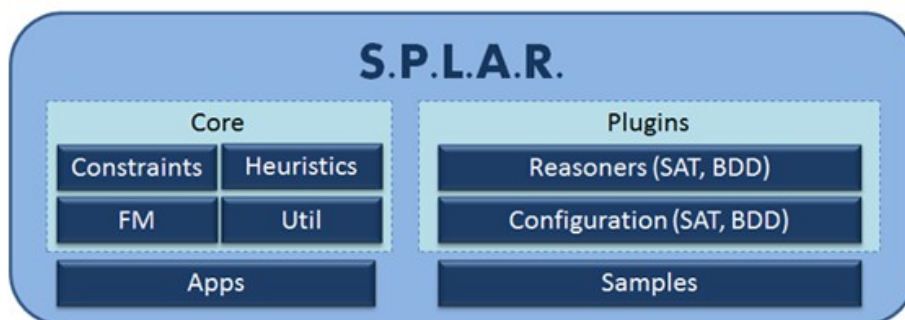
Figura 16: Arquitetura da Ferramenta



Fonte: elaborado pelo autor.

Como apresentado, a ferramenta proposta encontra-se em uma camada superior e utiliza-se do componente S.P.L.A.R. como uma biblioteca em um nível abaixo, tornando possível, além da leitura de arquivos que representam o modelo de *features*, especificamente do S.P.L.O.T, a estruturação do mesmo em artefatos menores que representam cada objeto, seja *feature* ou restrição de integridade, do modelo. Dessa forma, é possível aplicar as medidas definidas neste trabalho sobre os componentes do modelo conforme a estrutura definida.

Figura 17: S.P.L.A.R.



Fonte: S.P.L.O.T (2014)

Pode-se verificar na Figura 17 os componentes que fazem parte do S.P.L.A.R. No âmbito do *core*, há artefatos que representam e implementam as restrições de integridade do modelo de *feature*, o próprio modelo, heurísticas e utilidades que tratam a cerca do próprio do modelo. Em *plugins*, encontramos implementações e facilitadores para aplicação do SAT e BDD.

Na representação do modelo, há itens que representam cada *feature*, além de suas características, como ser mandatória ou optativa, podendo ser um agrupamento ou uma *feature* agrupada. Por outro lado, há nas restrições de integridade cada representação das formulas que a definem, contendo as *features* que participam e sua valoração na restrição. Já em heurísticas, há diversos algoritmos implementados de forma a facilitar a solução de problemas em análise de modelo de *features*.

Com SAT e BDD é possível aplicar algumas medidas em modelos de *features*, tais como checar a satisfabilidade do modelo, se há alguma *feature* morta, computar a quantidade de configurações válidas que podem ser geradas pelo modelo, entre outras. A Tabela 5 abaixo mostra algumas recomendações de aplicação com SAT e BDD.

Tabela 5: Recomendação de aplicação para SAT e BDD.

Aplicação	Caso
SAT	Checar a satisfabilidade do modelo
	Detectar <i>features</i> mortas no modelo
	Detectar se uma determinada <i>feature</i> está morta no modelo
	Verificar a equivalência entre modelos
	Checar a especialização de modelos
	Validar configuração de modelo
	Enumerar as configurações de modelo
BDD	Detectar <i>features</i> mortas no modelo
	Verificar a equivalência entre modelos
	Checar a especialização de modelos
	Calcular domínios válidos
	Enumerar as configurações de modelo
	Calcular o número de configurações válidas
	Calcular o fator de variabilidade
Calcular a comunalidade de uma <i>feature</i>	

Fonte: MENDONÇA, 2009.

4.2 Desenvolvimento

Após ter sido definida a modelagem da ferramenta, definindo seus requisitos funcionais e sua arquitetura, tomando como base um componente externo para auxílio na aplicação de medidas, foi realizado o desenvolvimento da mesma. No processo de desenvolvimento, fizeram parte a tarefa de importar os modelos de *features* de ferramentas

externas para a proposta neste trabalho, o desenvolvimento da área de visualização e edição do modelo já estruturado na ferramenta, além de possibilitar a exportação dos dados resultantes da análise de qualidade com as medidas implementadas. Segue na Figura 18, três exemplos de medidas implementadas na ferramenta: a complexidade composta de um modelo de *feature*, restrições de integridade e seu coeficiente de densidade de conectividade.

Figura 18: Exemplo de três medidas implementadas.

```

400 @Override
401 public double compoundComplexity() {
402
403     int features = featureModelStatistics.countFeatures();
404     int mandatoryFeatures = featureModelStatistics.countMandatory();
405     int orFeatures = featureModelStatistics.countGroups1N();
406     int xorFeatures = featureModelStatistics.countGroups11();
407     int groups = orFeatures + xorFeatures;
408     int constraints = featureModelStatistics.countConstraints();
409
410     int allRelationship = mandatoryFeatures + orFeatures + xorFeatures +
411
412     double result = Math.pow(features, 2) + (Math.pow(mandatoryFeatures,
413
414     return result;
415 }
416
417 @Override
418 public int crossTreeConstraints() {
419
420     return featureModelStatistics.countConstraints();
421 }
422
423 @Override
424 public double coefficientOfConnectivityDensity() {
425
426     if ( featureModel.countConstraints() == 0 )
427         return 0;
428
429     CNFFormula cnf = new CNFFormula();
430     boolean canAdd = true;
431
432     for( PropositionalFormula pf : featureModel.getConstraints() ) {
433         for(Resolution resolution : currentContext.getResolutions()){
434             if(pf.getVariable(resolution.getIdFeature()) != null && !resc
435                 canAdd = false;
436         }
437
438         if(canAdd)
439             cnf.addClauses(pf.toCNFClauses());
440     }
441
442     return (cnf == null) ? 0 : cnf.getClauseDensity();
443
444 }

```

Fonte: elaborada pelo autor.

4.2.1 Importação

O modelo de *feature* que a ferramenta trabalha segue a estrutura relativa ao modelo que a ferramenta S.P.L.O.T. gera. No caso, a estrutura é modelada conforme a tecnologia XML (BRAY et al., 1998), e a ferramenta faz a leitura da estrutura, separando-a em componentes que as representem e possa ser utilizada pela aplicação (Ver Figura 19).

Figura 19: Estrutura representada pelo XML do S.P.L.O.T.

```

1 <feature_model name="Mobile Phone">
2 <meta>
3 <data name="description">Nothing</data>
4 <data name="creator">AE</data>
5 <data name="address"></data>
6 <data name="email">alireza_ensan@yahoo.com</data>
7 <data name="phone"></data>
8 <data name="website"></data>
9 <data name="organization">UNB</data>
10 <data name="department"></data>
11 <data name="date"></data>
12 <data name="reference"></data>
13 </meta>
14 <feature_tree>
15 r Mobile Phone(_r)
16 m Utilit Functions(_r_1)
    i. m Calls(_r_1_2)
        1. :g(_r_1_2_7) [1,*]
            a. :Voice(_r_1_2_7_8)
            b. :Data(_r_1_2_7_9)
        ii. m Messaging(_r_1_3)
            1. :g(_r_1_3_10) [1,*]
                a. :SMS(_r_1_3_10_11)
                b. :EMS(_r_1_3_10_12)
                c. :MMS(_r_1_3_10_13)
        iii. :o Games(_r_1_4)
        iv. m Alarm clock(_r_1_5)
        v. m Ringing Tones(_r_1_6)
17 m Setting(_r_14)
    i. :o Java Support(_r_14_15)
    ii. m OS(_r_14_16)
        1. :g(_r_14_16_17) [1,1]
            a. :Symbian(_r_14_16_17_18)
            b. :WinCE(_r_14_16_17_19)
18 :o Media(_r_20)
    i. :o Camera(_r_20_21)
    ii. :o MP3(_r_20_22)
    iii. :o MP4(_r_20_23)
19 :o Connectivity(_r_24)
    i. :g(_r_24_25) [1,*]
        1. :Bluetooth(_r_24_25_26)
        2. :USB(_r_24_25_27)
        3. :Wifi(_r_24_25_28)
20 </feature_tree>
21 <constraints>
22 constraint_4:~_r_1_4 or _r_14_15
23 constraint_5:~_r_20_22 or ~_r_20_23
24 </constraints>
25 </feature_model>



















```

Fonte: S.P.L.O.T (2014)

Pode-se observar que, para a estrutura apresentada, foi necessário analisar como sua estrutura poderia ser importada para ser utilizada na atual ferramenta. Dessa forma, cada

representação encontrada no arquivo foi transposta para um artefato que a representasse na ferramenta, como segue na Figura 20:

Figura 20: Representação da estrutura do modelo.

- ▷  FeatureGroup.java
- ▷  FeatureModel.java
- ▷  FeatureModelException.java
- ▷  FeatureModelListener.java
- ▷  FeatureModelState.java
- ▷  FeatureModelStatistics.java
- ▷  FeatureTree.java
- ▷  FeatureTreeCellRenderer.java
- ▷  FeatureTreeNode.java
- ▷  FeatureTreeNodeState.java
- ▷  FeatureValueAssignmentException.java
- ▷  FTTraversalNodeSelector.java
- ▷  FTTraversals.java
- ▷  GroupedFeature.java
- ▷  IFNodeRenderer.java
- ▷  NodeRenderer.java
- ▷  RootNode.java
- ▷  SolitaireFeature.java

Fonte: elaborada pelo autor.

O componente “*FeatureModel*” representa toda a estrutura do modelo de *feature* importada para a ferramenta. Já para representar as *features*, há os componentes “*FeatureGroup*”, “*GroupedFeature*”, “*RootNode*” e “*SolitaireFeature*”, que representam as *features* como grupos, *features* filhas de grupos, *feature* raiz e *features* mandatórias ou opcionais que não são grupos ou filhas de grupos, respectivamente.

Contudo, a estrutura anteriormente citada não possui representações de contexto, sendo necessária a criação de novos componentes para reproduzir os mesmos. Assim, foi criado um componente que englobasse a estrutura anterior e possuísse as estruturas de contexto. Segue a Figura 21 abaixo com a representação de contexto:

Figura 21: Componente de representação do modelo com contextos.

```

38 public class ContextModel implements IContextModel {
39
40     private FeatureModel featureModel;
41     private ReasoningWithBDD bddReasoner;
42     private ReasoningWithSAT satReasoner;
43     private FeatureModelStatistics featureModelStatistics;
44
45     private String pathModelFile;
46
47     /*
48      * Context-Aware
49      */
50
51     private Map<String, Context> contexts;
52     private Map<Context, FeatureModelStatistics> statisticsByContext;
53     private Map<Context, ReasoningWithBDD> bddByContext;
54     private Context currentContext;
55     private String modelName;
56

```

Fonte: elaborada pelo autor.

Com o novo componente é possível representar a estrutura do modelo de *feature*, bem como todos os contextos, permitindo que a análise de qualidade com a aplicação das medidas seja feita de forma adequada, aplicando-as conforme o contexto apropriado.

4.2.2 Área de visualização

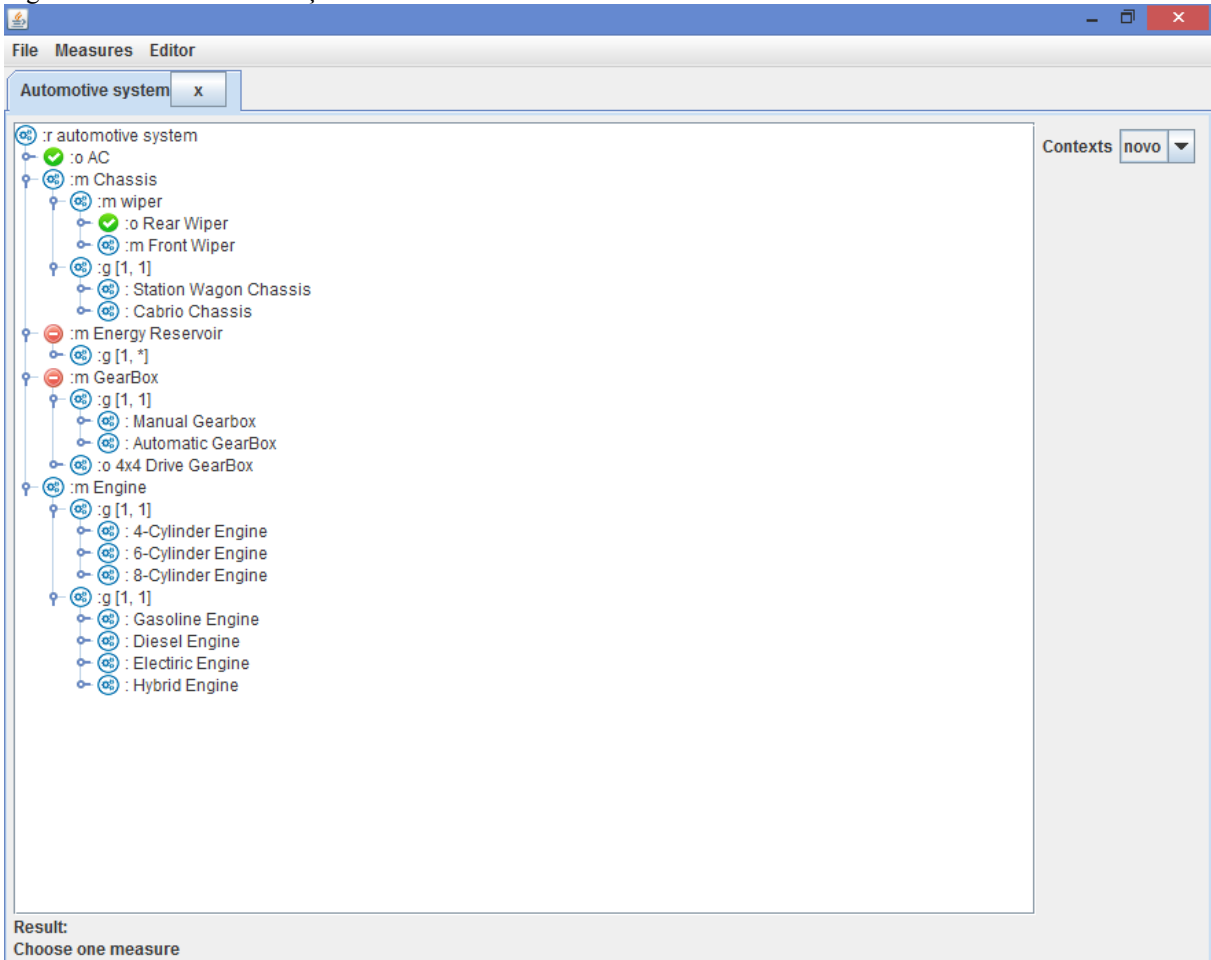
Conforme apresentado na Figura 22, verifica-se que há uma área específica para exibir o modelo de *feature* e outra para exibir informações a respeito da mesma. Na área de informação, podemos constatar a presença de um componente que lista todos os contextos presentes no modelo, possibilitando ao usuário a escolha de um contexto específico e, automaticamente, atualizará o modelo visível para representar sua estrutura conforme tal contexto.

De informações adicionais, na parte inferior da tela, observa-se um espaço que exibe a medida escolhida, conforme as diversas possíveis apresentadas no *menu* medidas da aplicação, bem como o resultado desta.

Podemos perceber na árvore de apresentação do modelo as *features* presentes, bem como seus ícones indicadores, representando as *features* que estão ativas para o determinado contexto escolhido, bem como também as *features* que não estão ativas, além das que realmente não participam de tal contexto.

Dessa forma, a área de visualização possibilita uma análise breve sobre a estrutura do modelo referente a um determinado contexto, como também permite aplicar medidas individuais para uma percepção rápida sobre os resultados.

Figura 22: Área de visualização

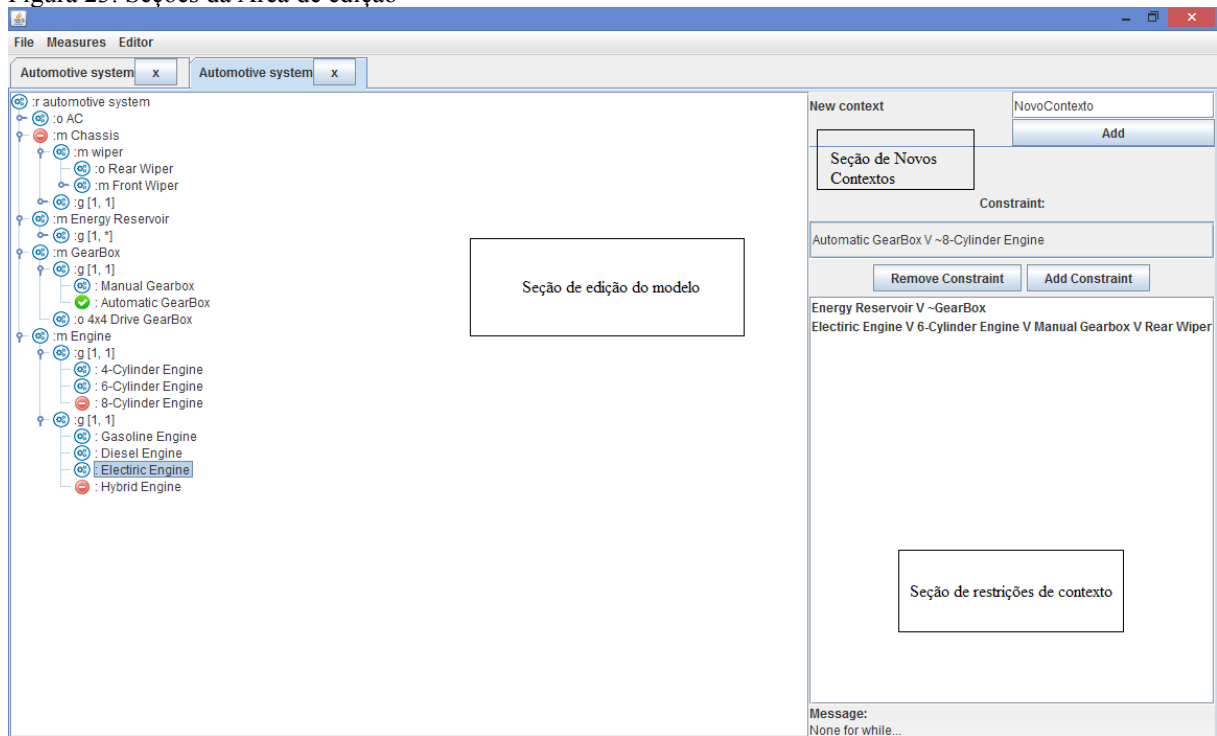


Fonte: elaborada pelo autor.

4.2.3 Área de edição

Para a área de edição foi definido três subseções: (i) seção de edição do modelo; (ii) adicionar o novo contexto; e (iii) adicionar novas restrições de integridade de contexto, de acordo com a Figura 23.

Figura 23: Seções da Área de edição



Fonte: elaborada pelo autor.

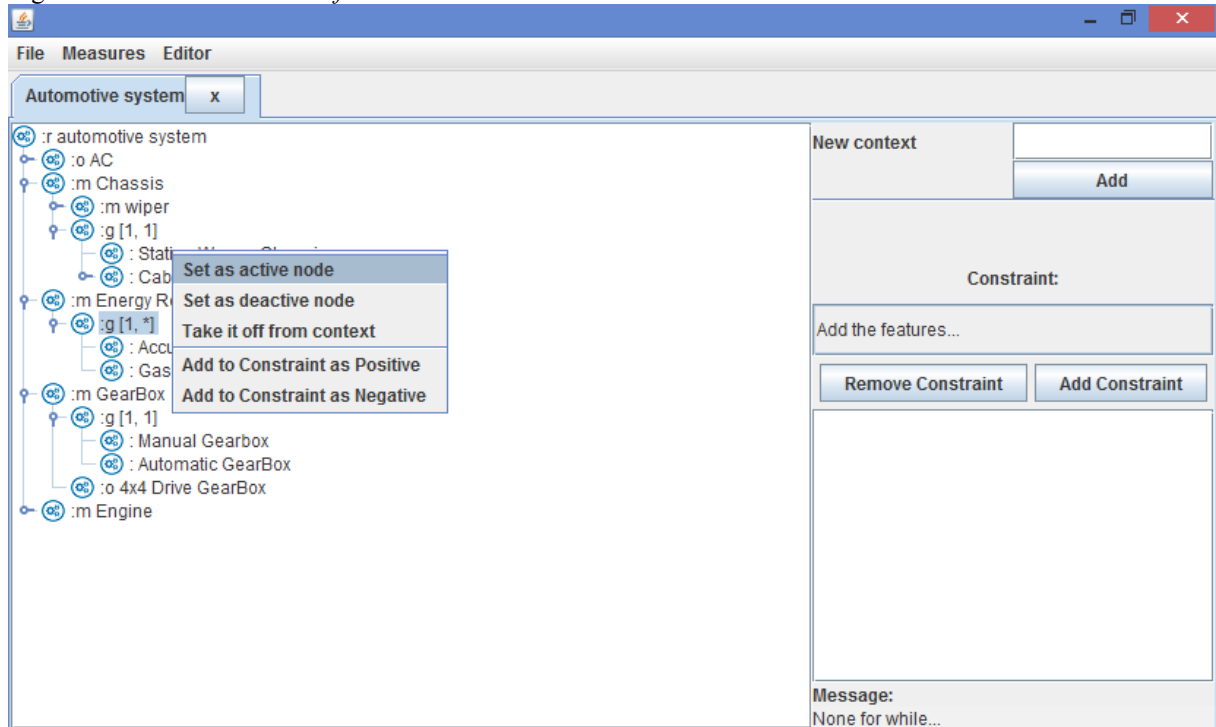
Na Seção de edição do modelo podemos definir uma *feature* como uma característica ativa ou não ativa no contexto que estamos criando, além de realmente podermos eliminá-la de tal contexto, caso ela já tenha sido inserida, por exemplo. Também, no mesmo menu, disponível ao clicar com o botão direito do mouse sobre uma determinada *feature*, permitenos adicionar tal *feature* às regras de restrições, assumindo seu estado positivo ou negativo na fórmula (Ver Figura 24). Dessa forma, é possível definir quais *features* participam do contexto e quais estão ativas ou não nesse contexto, representando como se estivesse em um determinado ambiente para determinadas informações de contexto.

Em relação à seção de restrições de contexto, podemos visualizar a atual restrição que está sendo criada, conforme o usuário adiciona as *features* apresentadas, adicionar tal restrição de contexto em criação, como também remover a restrição de contexto, seja a que está em andamento na sua criação ou a que já foi adicionada e consta na lista das restrições pertencentes ao contexto em edição. Para tal ação de remoção, segue-se o mesmo princípio de menu disponível na área de edição do modelo, clicando com o botão direito do mouse sobre a restrição que deseja remover e seleciona a ação de remoção.

Por fim, apresenta-se a seção de novos contextos, consistindo apenas de uma área onde é possível adicionar um nome ao atual contexto em edição e permitindo salvar tal

contexto ao modelo de *feature*, conseqüentemente, salvando todos os dados em formato XML (BRAY, 1998) obedecendo à estrutura do modelo gerado pela ferramenta a qual se exportou tal modelo.

Figura 24: Ativar ou desativar *features*



Fonte: elaborada pelo autor.

4.2.4 Exportação

Por último, foi importante desenvolver uma forma de exportar os resultados das medidas de qualidade para que as mesmas fossem analisadas por uma ferramenta externa e que pudesse oferecer as funcionalidades adequadas. Assim, a ferramenta atual exporta os resultados para o formato que a aplicação Office Excel (LAPPONI, 2005) trabalhe, possibilitando análises estatísticas e geração de gráficos.

Figura 25: Imagens do Excel com os dados exportados.

		Modelo exportado		Contexto	
	A	B	C	D	
1	Measures	Mobile-Phone (only-music)	Mobile-Phone (down-battery)	Mobile-Phone (high-battery)	
2	Non-Functional Commonality (NFC)	0	0	0	0
3	Number of features (NF)	7	8	10	
4	Number of top features (NTop)	2	3	4	
5	Number of leaf Features (NLeaf)	4	5	7	
6	Depth of tree (DT)	3	3	3	
7	Cognitive Complexity of a Feature Model (I)	2	2	2	
8	Feature EXtendibility (FEX)	4	5	7	
9	Flexibility of configuration (FoC)	0,142857143	0,125	0,2	
10	Single Cyclic Dependent Features (SCDF)	0	0	0	
11	Multiple Cyclic Dependent Features (MCDF)	0	0	0	
12	Cyclomatic complexity	2	2	2	
13	Compound Complexity	67,33333333	87,66666667	123,6666667	
14	Cross-tree constraints (CTC)	2	2	2	
15	Coefficient of connectivity-density (CoC)	0	0	0,5	
16	Number of variable features	4	4	5	
17	Number of variation points	2	2	2	
18	Single Hotspot Features	3	3	3	
19	Multiple Hotspot Features	1	1	2	
20	Rigid Nohotspot Features	3	4	5	
21	Ratio of variability (RoV)	0,046875	0,0234375	0,013671875	
22	Number of valid configurations (NVC)	6	6	14	
23	Number of Activated Features	2	1	2	
24	Number of Deactivated Features	3	2	0	
25	Number of Context Constraints	0	2	1	

Fonte: elaborado pelo autor.

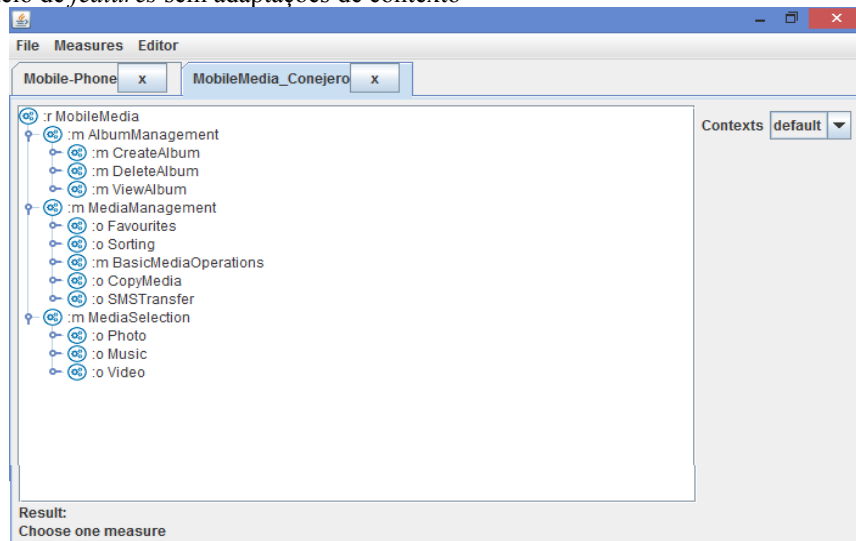
5 AVALIAÇÃO DO USO DA FERRAMENTA

Nesta Seção, será apresentada a avaliação do uso da ferramenta em relação à correteude da coleta das medidas e realizada uma análise da correlação das medidas para trinta modelos de *features* extraídos da ferramenta S.P.L.O.T.

5.1 Avaliação da Coleta das Medidas

Na ferramenta, são disponibilizadas três funcionalidades essenciais: visualizar um modelo com ou sem contexto, aplicando-se as medidas implementadas; editar um modelo sem ou com contextos, adicionando novos contextos conforme ativação e desativação de *features* e adição de novas regras de restrição; e exportar todos os resultados das medidas aplicados em modelos com ou sem contexto.

Para validar as medidas implementadas na ferramenta, será utilizado um modelo de *features* simples sem adaptações de contexto (Ver Figura 26).

Figura 26: Modelo de *features* sem adaptações de contexto

Fonte: elaborado pelo autor.

Dessa forma foi realizada a exportação de Excel com a coleta de todas as medidas referentes ao modelo de *features* (Ver Figura 27). Assim, foi realizada a coleta manual das medidas e comparada com a coleta automática da Figura 27. Foi verificado a corretude dos resultados das medidas.

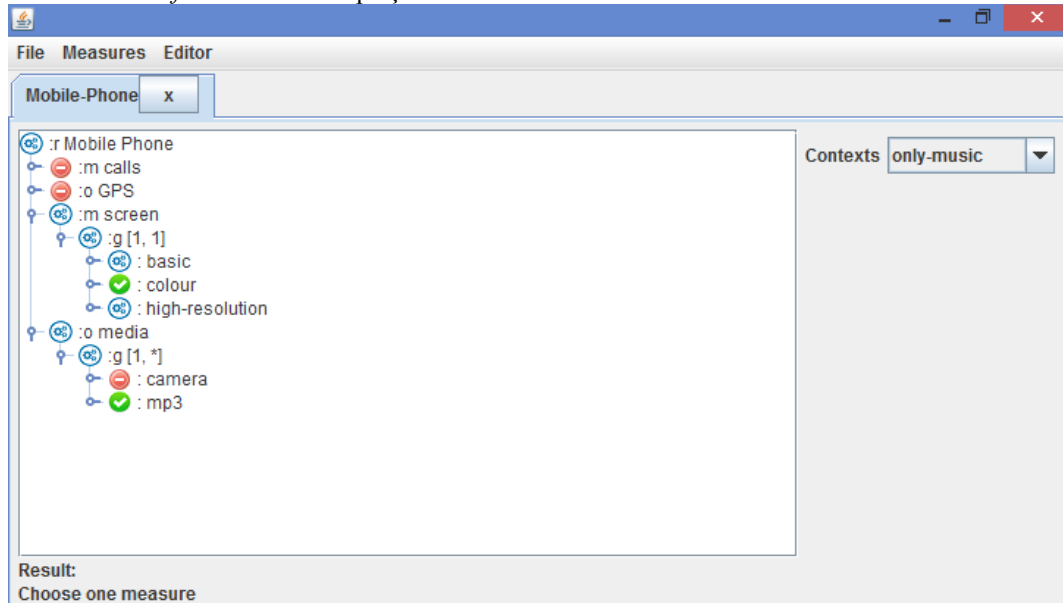
Figura 27: Validação exportação de modelo sem contexto

	A	B
1	Measures	Mobile-Phone
2	Non-Functional Commonality (NFC)	0
3	Number of features (NF)	10
4	Number of top features (NTop)	4
5	Number of leaf Features (NLeaf)	7
6	Depth of tree (DT)	3
7	Cognitive Complexity of a Feature Model (I)	2
8	Feature EXTendibility (FEX)	7
9	Flexibility of configuration (FoC)	0,2
10	Single Cyclic Dependent Features (SCDF)	0
11	Multiple Cyclic Dependent Features (MCDF)	0
12	Cyclomatic complexity	2
13	Compound Complexity	123,6666667
14	Cross-tree constraints (CTC)	2
15	Coefficient of connectivity-density (CoC)	0,5
16	Number of variable features	5
17	Number of variation points	2
18	Single Hotspot Features	3
19	Multiple Hotspot Features	2
20	Rigid Nohotspot Features	5
21	Ratio of variability (RoV)	0,013671875
22	Number of valid configurations (NVC)	14
23	Number of Contexts	8

Fonte: elaborado pelo autor

Também foram analisadas as medidas de um modelo de *features* com adaptações de contexto (Ver Figura 28).

Figura 28: Modelo de *features* com adaptações de contexto.



Fonte: elaborado pelo autor

Da mesma forma que no modelo de *features* sem contexto, foi analisada as medidas para cada contexto. Dessa forma, novas *features* são adicionadas: Número de *features* ativadas, Número de *features* desativadas e Número de restrições de contextos. Verifica-se também que as medidas relacionadas às LPSs, mudam de acordo com o contexto relacionado. Pode-se observar, por exemplo, na Figura 29 a medida que indica o Número de *Features* Ativas para o contexto “*only-music*”, representando um valor que realmente consiste com a estrutura criada conforme o modelo de exemplo utilizado (Ver Figura 28).

Figura 29: Validação exportação de modelo com contexto

	A	B	C	D
1	Measures	Mobile-Phone (only-music)	Mobile-Phone (down-battery)	Mobile-Phone (high-battery)
2	Non-Functional Commonality (NFC)	0	0	0
3	Number of features (NF)	7	8	10
4	Number of top features (NTop)	2	3	4
5	Number of leaf Features (NLeaf)	4	5	7
6	Depth of tree (DT)	3	3	3
7	Cognitive Complexity of a Feature Model (I)	2	2	2
8	Feature EXTendibility (FEX)	4	5	7
9	Flexibility of configuration (FoC)	0,142857143	0,125	0,2
10	Single Cyclic Dependent Features (SCDF)	0	0	0
11	Multiple Cyclic Dependent Features (MCDF)	0	0	0
12	Cyclomatic complexity	2	2	2
13	Compound Complexity	67,33333333	87,66666667	123,6666667
14	Cross-tree constraints (CTC)	2	2	2
15	Coefficient of connectivity-density (CoC)	0	0	0,5
16	Number of variable features	4	4	5
17	Number of variation points	2	2	2
18	Single Hotspot Features	3	3	3
19	Multiple Hotspot Features	1	1	2
20	Rigid Nohotspot Features	3	4	5
21	Ratio of variability (RoV)	0,046875	0,0234375	0,013671875
22	Number of valid configurations (NVC)	6	6	14
23	Number of Activated Features	2	1	2
24	Number of Deactivated Features	3	2	0
25	Number of Context Constraints	0	2	1

Fonte: elaborado pelo autor

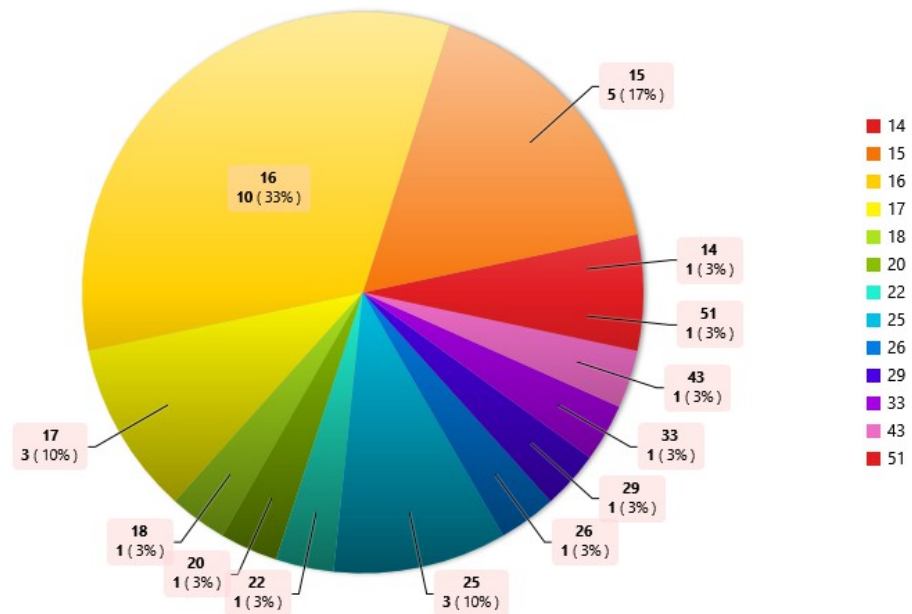
Outras medidas podem ser facilmente verificadas em Figura 27 e Figura 29 quanto ao seu valor em relação à estrutura apresentada, tais como o Número de Pontos Variantes, o Número de *Features* Folha, Número de *Features* Desativadas, *Features* Top e a Profundidade da árvore, por exemplo.

Dessa forma, pode-se verificar que o cálculo das medidas está sendo realizado de forma correta. No entanto, é necessário avaliar o uso da ferramenta por especialistas em LPSs para poder obter melhores resultados para validação da ferramenta.

5.2 Análise de Correlação das Medidas

Para realizar a avaliação, foram avaliados trinta modelos, aplicando-se todas as medidas e exportando seus resultados, conforme funcionalidade implementada na ferramenta, para serem analisadas pelo *software* Epi Info® (EPI-INFO, 1997). Para delimitar tais modelos, o domínio de aplicação abrangido foi referente à palavra-chave “Mobile”. Contudo, todos os modelos constavam no repositório da plataforma S.P.L.O.T. (S.P.L.O.T., 2014).

Figura 30: Quantidade de *features* dos modelos selecionados do SPLOT.

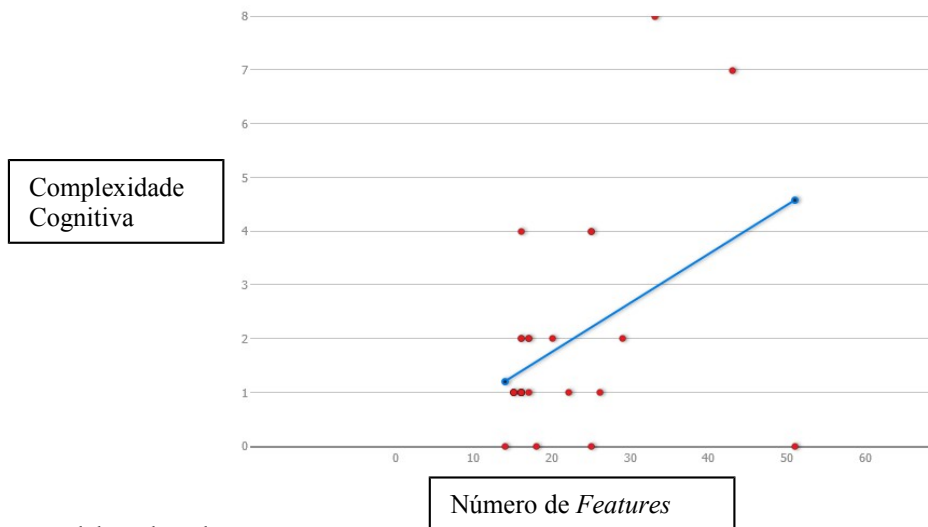


Fonte: elaborada pelo autor

Conforme Figura 30, observa-se que, com base nos 30 modelos analisados, grande parte possuíam dezesseis *features*, representando cerca de 33% dos modelos totais. A partir dessa análise inicial, aplicou-se comparações entre os resultados obtidos com a ferramenta, buscando comprovar que a implementação das medidas da ferramenta possam estar coesas. As análises de correlações das medidas foram realizadas com o intervalo de confiança de 95%.

Com isso, o primeiro fator observado foi entre as medidas Número de *Features* e Complexidade Cognitiva (Ver Figura 31). Podemos perceber que há uma alta dispersão entre os resultados, concluindo-se que não há correlação entre as variáveis expostas, tornando verdade o significado de suas relações, pois, dado que a Complexidade Cognitiva representa o quão fácil pode ser entender um software em relação a sua variabilidade, é lógico que isso pode independer do número de *features* presentes em um modelo. No entanto, se apresentasse correlação, seria uma correlação positiva, pois ambas tendem ou crescer ou a diminuir juntas.

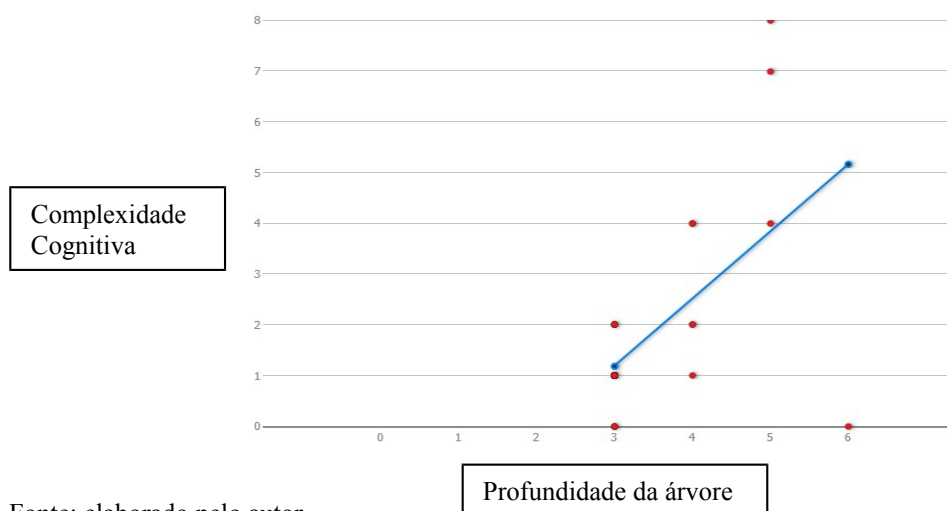
Figura 31: Análise com Número de Features x Complexidade Cognitiva



Fonte: elaborado pelo autor

Para a segunda análise, as medidas observadas foram Profundidade da árvore de modelo e Complexidade Cognitiva (Ver Figura 32). Podemos observar que poderiam possuir uma correlação positiva, porém devido à dispersão observada, também não há correlação. Essa análise segue o mesmo exemplo da anterior, dado que a complexidade referente à variabilidade do modelo pode variar independentemente da profundidade da árvore.

Figura 32: Análise com Profundidade da árvore de modelo x Complexidade Cognitiva

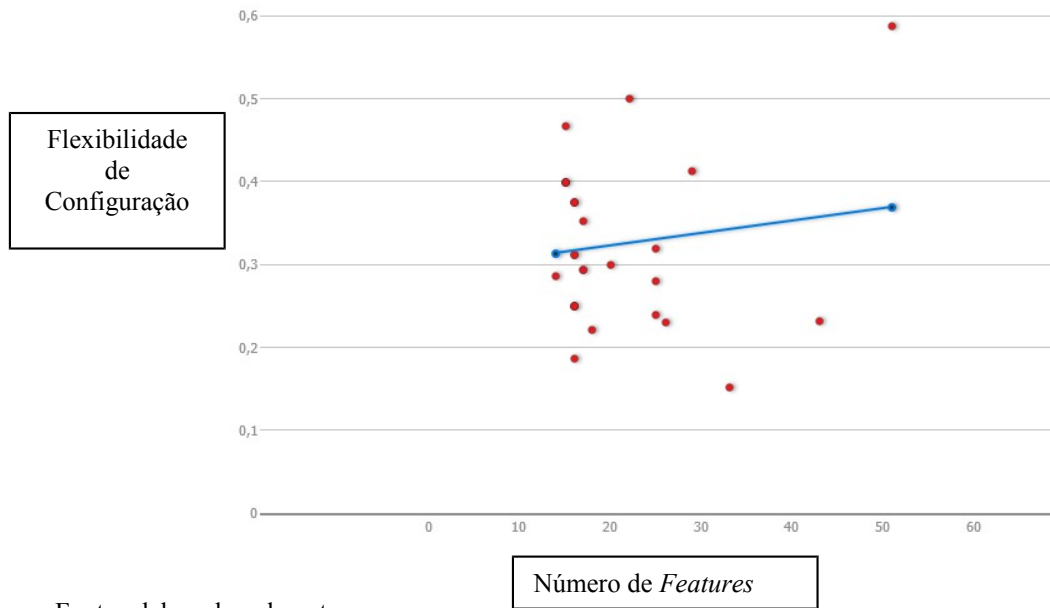


Fonte: elaborado pelo autor

Outra observação importante foi em relação ao Número de *Features* e Flexibilidade de Configuração, conforme Figura 33. Podemos notar uma diferença em tal análise, pois a dispersão encontrada é menor em relação às análises anterior, indicando que pode haver correlação positiva entre as variáveis. Comparando-se as variáveis, percebemos que realmente

há relação, pois a Flexibilidade de Configuração, denotando a possibilidade de gerar configurações distintas, é dependente dos valores do Número de *Features*, dado que esta última pode contribuir para aumentar ou diminuir os resultados de Flexibilidade de Configuração.

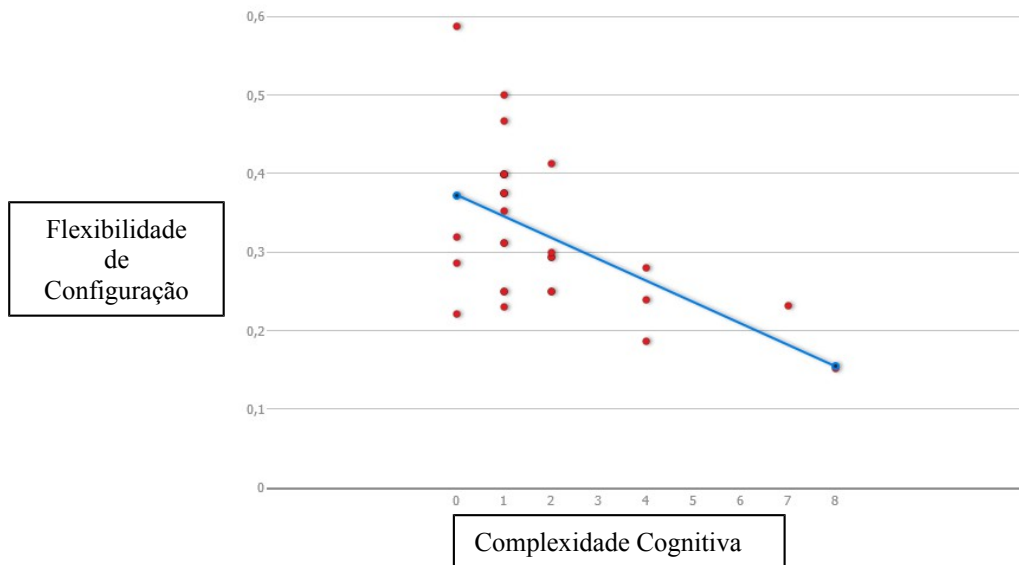
Figura 33: Análise com Número de *Features* x Flexibilidade de Configuração



Fonte: elaborado pelo autor

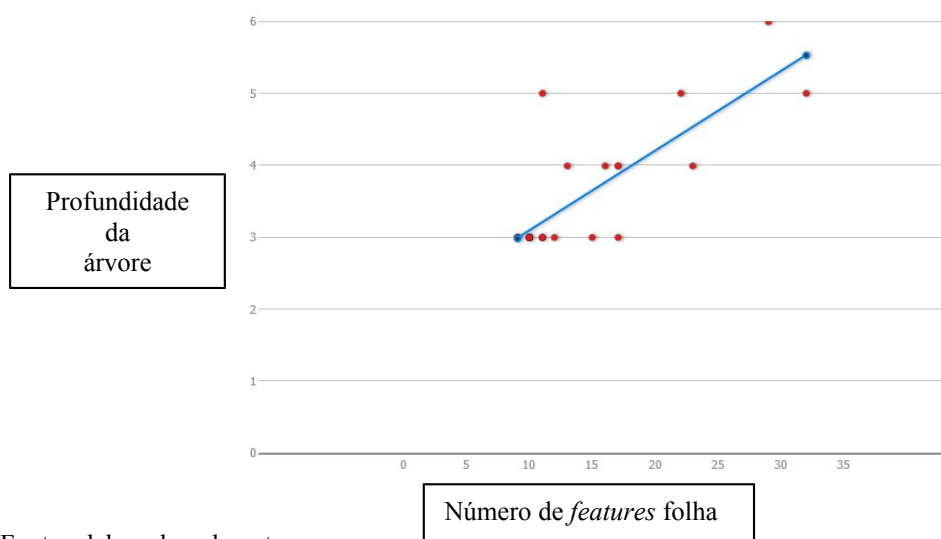
Dado as duas medidas bases analisadas anteriormente, a próxima observação foi utilizando-as, sendo uma análise entre as medidas Complexidade Cognitiva e Flexibilidade de Configuração (Ver Figura 34). Conseguimos visualizar no gráfico que houve um resultado esperado, dado que constatamos uma possível correlação negativa, representando que cada medida pode variar em seus valores isoladamente da outra. Pois, seguindo o significado das medidas observadas, a possibilidade de gerar configurações distintas no modelo independe do modelo ser facilmente entendido.

Figura 34: Análise com Complexidade Cognitiva x Flexibilidade de Configuração



Fonte: elaborado pelo autor.

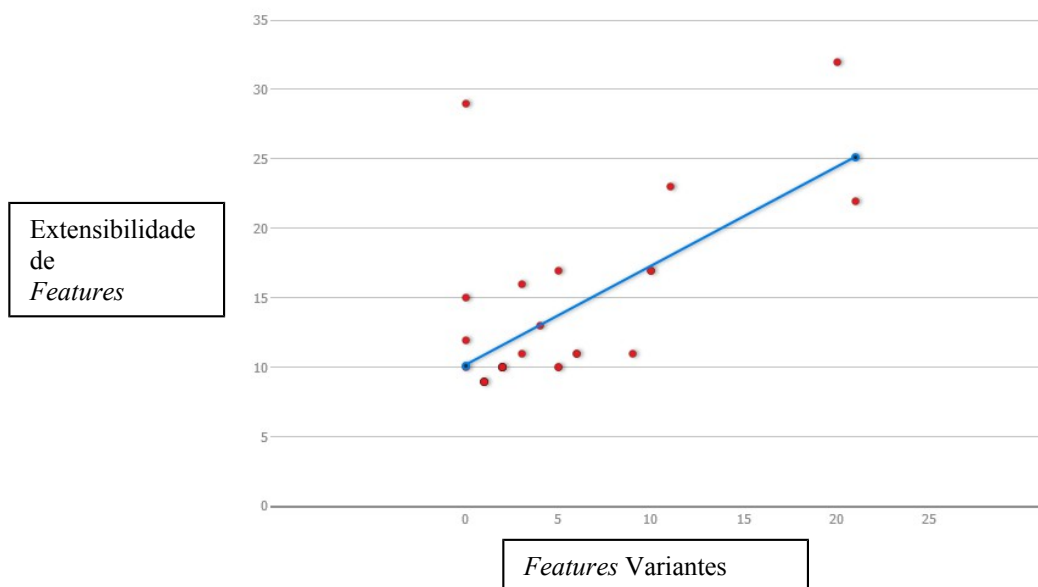
Outra correlação interessante a ser analisada foi entre as medidas de Número de *Features* Folha e Profundidade da Árvore. Percebe-se que há uma possível correlação positiva, conforme Figura 35, significando verdade a relação que ambas possuem, pois as *features* folhas geram ramificações, o que pode definir uma maior profundidade à árvore do modelo.

Figura 35: Análise com Número de *features* folha x Profundidade da Árvore

Fonte: elaborado pelo autor

Por último, observaram-se as medidas *Features* Variantes e Extensibilidade de *Features*, segundo Figura 36. Com relação aos significados, Extensibilidade de *Feature* representa as *features* que podem ser facilmente adicionadas em tempo de manutenção, o que abrange comumente *features* sem filhos, tais como *features* folhas ou mesmo as filhas de agrupamentos. E *Features* Variantes denota *features* filhas de agrupamentos, independente da cardinalidade apresentada. Assim, constamos na análise uma possível correlação positiva, pois ambas tendem a variar de forma semelhante, seja crescer ou diminuir em seus valores, dado que, conforme seus significados apresentados, realmente possuem relações.

Figura 36: Análise com *Features* Variantes x Extensibilidade de *Features*



Fonte: elaborado pelo autor

Conforme as análises desenvolvidas, podemos observar relações entre as medidas contempladas, como também uma maneira de tentar demonstrar a corretude de tais medidas implementadas na ferramenta proposta neste trabalho. Porém, outras análises que incluam medidas relacionadas modelos de *features* com adaptações de contextos precisam ser realizadas, pois não foi possível desenvolver neste trabalho devido à limitação dos modelos de *features* que abrangessem adaptações de contextos, impossibilitando a geração de dados.

6 CONSIDERAÇÕES FINAIS

LPS é uma abordagem de interesse das organizações de software pelas suas características e possíveis retornos, pois a sua essência é a reutilização, permitindo a redução

de custos e maior produtividade. Contudo, a sua utilização torna-se ineficaz em relação aos sistemas atuais em desenvolvimento, os sistemas sensíveis ao contexto, que necessitam de adaptações e geração de novas configurações em tempo de execução. Dessa forma, as Linhas de Produto de Software Dinâmicas buscam suprir tal ineficiência, possibilitando que os requisitos e restrições de ambiente possam ser ajustados e mantidos conforme necessários. No entanto, a análise de qualidade torna-se uma tarefa essencial para o desenvolvimento das LPSs, seja as tradicionais ou as dinâmicas, pois busca garantir o propósito do desenvolvimento com qualidade, evitando que erros possam ser dominantes e propagados. Assim, uma ferramenta foi proposta a avaliação da qualidade do modelo de *features* em Linhas de Produto de Softwares Dinâmicas.

Inicialmente, ocorreu um levantamento de ferramentas que trabalhassem com LPS e, posteriormente, tais ferramentas foram analisadas quanto à sua estrutura e funcionalidades. A partir de tais estudos, foi possível fazer o levantamento de requisitos funcionais básicos, que podiam fazer parte também da ferramenta proposta neste trabalho e, conseqüentemente, uma análise foi realizada para definir funcionalidades que abrangessem questões em LPSD.

As deficiências apresentadas pelas ferramentas levantadas foram desenvolvidas a partir da nova ferramenta. Tais funcionalidades possibilita a visualização de modelos de *feature* com inserção de adaptações de contextos, aplicando medidas implementadas na própria ferramenta; a edição de modelos de LPS tradicionais para que adaptações de contextos possam ser adicionados, bem como suas restrições de integridade; além de permitir a exportação dos dados de análise dos modelos, seja ele com ou sem contexto, apresentando todas as medidas correspondentes com seus respectivos resultados.

Após o desenvolvimento, foi realizada a avaliação final da ferramenta, verificando todos os requisitos levantados e o seu funcionamento, além de produzir uma análise de resultados a partir da aplicação das medidas implementadas, desenvolvendo-se a geração de gráficos e comparação dos resultados.

Com o trabalho foi possível perceber que a LPS, mesmo sendo ineficaz quanto aos sistemas sensíveis ao contexto em desenvolvimento, ainda é a abordagem mais utilizada no que tange a implementação de ferramentas para análise e garantia de qualidade de tais artefatos. Ou mesmo, as ferramentas disponibilizadas não fornecem uma grande quantidade de medidas, diminuindo as oportunidades e resultados para uma análise de qualidade. Por isso, evidenciou-se a necessidade de desenvolver uma ferramenta que pudesse apoiar na

análise de modelos em LPSD e também na garantia de sua qualidade, apresentando medidas não antes abordadas por outras ferramentas.

Contudo, a ferramenta proposta limita-se a interpretar modelos advindos apenas da ferramenta S.P.L.O.T. Além disso, há características importantes em sistemas sensíveis ao contexto que podem fazer parte dos modelos em LPSD que não foram contempladas, como a criação de modelos de *features* estendido, permitindo que haja valoração em *features* conforme contexto definido (Benavides et al, 2005; Kang et al, 1998; Czarnecki et al, 2005). Então, em trabalhos futuros, pretende-se expandir a ferramenta para interpretar mais estruturas de modelos de *features* de outras ferramentas, além de desenvolver a abordagem a de modelos de *features* estendidos.

Também é pretendido incorporar outras medidas à ferramenta para a avaliação de LPSDs. Por fim, pretende-se também realizar uma avaliação da usabilidade da ferramenta por usuários especialistas em LPSs.

REFERÊNCIAS

- ABOWD, G. D. et al. Towards a better understanding of context and context-awareness. In: *Handheld and ubiquitous computing*. Springer Berlin Heidelberg, p. 304-307, 1999.
- BENAVIDES, D.; SEGURA, S.; RUIZ-CORTÉS, A. Automated Analysis of *feature* models 20 years later: A Literature Review. **Information Systems Journal**, v.35, n.6, p. 615-636, set., 2010.
- BENAVIDES, D.; TRINIDAD, P.; RUIZ-CORTÉS, A. Automated reasoning on feature models. In: **17th Int. Conf. Advanced Information Systems Engineering**, Springer-Verlag, LNCS, vol 3520, pp 491-503. 2005.
- BENCOMO, N.; HALLSTEINSEN, S.; SANTANA DE ALMEIDA, E. A View of the Landscape of Dynamic Software Product Lines. **Computer**, v. 45, n. 10, p. 36-41, 2012.
- BEUCHE, D.; DALGARNO, M. Software Product Line Engineering with *Feature Models*. **Overload Journal**, v. 78, p. 5-8, abr., 2007. Disponível em: <<http://www.accu.org/var/uploads/journals/Overload78.pdf>>. Acesso em: 13 ago. 2014
- BEZERRA, C. I. M.; ANDRADE, R. M. C.; MONTEIRO, J. M. S. Avaliação da Qualidade do Modelo de *Features* em Linhas de Produtos de Software Utilizando Medidas. XII Simpósio Brasileiro De Qualidade De Software, 2013.
- BEZERRA, C. I. M.; ANDRADE, R. M. C.; MONTEIRO, J. M. S. Measures for Quality Evaluation of Feature Models. 14th International Conference on Software Reuse. Miami, USA. Springer, 2015. (artigo aceito para publicação)
- BRAY, T. et al. Extensible markup language (XML). World Wide Web Consortium Recommendation REC-xml-19980210. <http://www.w3.org/TR/1998/REC-xml-19980210>, 1998.
- BRYANT, R. E. Graph-based algorithms for boolean function manipulation. **Computers, IEEE Transactions on**, v. 100, n. 8, p. 677-691, 1986.
- BROWN, P. J.; BOVEY, J. D.; CHEN, X. **Context-aware applications: from the laboratory to the marketplace**. Personal Communications, IEEE, v. 4, n. 5, p. 58-64, 1997.
- BOUZID, C.; KRAIEM, N.; AL KHANJARI, Z. Towards a Dynamic Software Product Line: Analysis of the Background and State of the Art. **International Journal of Research in Business and Technology**, v. 4, n. 3, p. 489-501, 2014.
- CAPILLA, R. et al. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. **Journal of Systems and Software**, v. 91, p. 3-23, 2014.
- CAPILLA, R.; BOSCH, J. The promise and challenge of runtime variability. **Computer**, v. 44, n. 12, p. 93-95, 2011.

CAPILLA, R.; ORTIZ, O.; HINCHEY, M. Context Variability for Context-Aware Systems. **IEE Computer Society**, p. 85-87, 2014.

CELES, W.; DE FIGUEIREDO, L. H.; IERUSALIMSCHY, R. **A Linguagem Lua e suas Aplicações em Jogos**. Rio de Janeiro, 2004.

CZARNECKI, Krzysztof; HELSEN, Simon; EISENECKER, Ulrich. **Formalizing**

cardinality-based feature models and their specialization. **Software Process: Improvement and Practice**, v. 10, n. 1, p. 7-29, 2005.

DEY, A. K.; ABOWD, G. D.; SALBER, D. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. **Human-computer interaction**, v. 16, n. 2, p. 97-166, 2001.

DEY, A. K.; ABOWD, G. D.; SALBER, D. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. **Human-computer interaction**, v. 16, n. 2, p. 97-166, 2001.

DEY, A. K., ABOWD, G. D. Towards a Better Understanding of Context and Context-Awareness, In: **Proceedings of the CHI 2000 Workshop on The What, Who, Where, When, and How of Context-Awareness**, The Hague Netherlands. 2000.

DEY, A. K. Context-aware computing: The CyberDesk project. In: **Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments**. p. 51-54, 1998.

EPI-INFO 6.04. Center for disease control & prevention (CDC). WHO, 1997.

ETXE BERRIA, L.; SAGARDUI G.; BELATEGI, L. Quality aware Software Product Line Engineering. **Journal of the Brazilian Computer Society (JBACS)**, vol.14, no.1, Mar, 2008.

FAMILIAR. *Feature Model script* Language for manipulation and Automatic Reasoning. Disponível em: <<https://nyx.unice.fr/projects/familiar>>. Acesso em: 14 ago. 2014

FERNANDES, P. C. C. Ubifex: Uma Abordagem para Modelagem de Características de Linha de Produtos de Software Sensíveis ao Contexto. 2009. 126 f. Dissertação (Mestrado em Engenharia de Sistemas e Computação) – Universidade Federal do Rio de Janeiro. Rio de Janeiro, 2009.

FERNANDES, P.; WERNER, C.; MURTA L. *Feature Modeling for Context-Aware Software Product Lines*. In: **SEKE**. p. 758-763, 2008.

FRIEDMAN, Steven J.; SUPOWIT, Kenneth J. Finding the optimal variable ordering for binary decision diagrams. In: **Proceedings of the 24th ACM/IEEE Design Automation Conference**. **ACM**. p. 348-356, 1987

HALLSTEINSEN, S. et al. Dynamic software product lines. **Computer**, v. 41, n. 4, p. 93-95, 2008.

HTML. Tecnologia HTML. Disponível em: <http://www.w3schools.com/html/html_intro.asp>. Acesso em 25 set. 2014.

JAVA. Tecnologia JAVA. Disponível em:

<<http://www.oracle.com/technetwork/pt/java/javase/overview/index.html>>. Acesso em 25 set. 2014.

KANG, KC.; KIM, S.; LEE, J.; KIM, K.; SHIN, E.; HUH, M. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering* 5(1):143–168. 1998.

KJELDSKOV, J.; SKOV, M. B. Supporting work activities in healthcare by mobile electronic patient records. In: **Computer Human Interaction**. Springer Berlin Heidelberg, p. 191-200. 2004.

LAPPONI, J. C. Estatística usando excel. **Elsevier Brasil**, 2005.

MARINHO, F. G. PRECISE: Um Processo de Verificação Formal para Modelos de Características de Aplicações Móveis e Sensíveis ao Contexto. 2012. 182 f. Tese (Doutorado) - Curso de Ciência da Computação, Universidade Federal do Ceará, Fortaleza, 2012.

MENDONÇA, M.; WASOWSKI, A; CZARNECKI, K. SAT-based Analysis of *Feature Models* is Easy. In: XIII International Software Product Line Conference, 2009. Disponível em: <http://gsd.uwaterloo.ca:8088/SPLIT/articles/mendonca_sat_analysis_splc_2009.pdf>. Acesso em: 19 ago. 2014.

MENDONÇA, M. Efficient reasoning techniques for large scale *feature* models. Tese de Doutorado. University of Waterloo. 2009.

MONTAGUD, S.; ABRAHÃO, S. Gathering Current Knowledge about Quality Evaluation in Software Product Lines. In: **XIII International Software Product Line Conference (SPLC)**, San Francisco, USA, 2009.

MUNIR, Q.; SHAHID, M. Software Product Line: Survey of Tools. 2010. 60 f. Tese (Doutorado em Ciência da Computação) – Department of Computer and Information Science, Linköping universitet. Linköping, Suécia, 2010.

ROBAK, S. *Feature* modeling notations for system families. In: **International Workshop on Software Variability Management (SVM)**. p. 58. 2003.

ROSEMBERG, C. et al. Prototipação de software e design participativo: uma experiência do atlântico. In: **Proceedings of the VIII Brazilian Symposium on Human Factors in Computing Systems**. Sociedade Brasileira de Computação, 2008. p. 312-315.

SILVA, F. A. P. et al. Linhas de Produtos de Software: Uma tendência da indústria. In: IV Escola Regional de Informática: Ceará, Piauí e Maranhão, ERCEMAPI, 2010.

S.P.L.O.T. Software Product Line Online Tools. Disponível em: <<http://www.splot-research.org/>>. Acesso em 13 ago. 2014.

VIEIRA, V. et al. Uso e Representação de Contexto em Sistemas Computacionais. **Cesar AC Teixeira; Clever Ricardo G. de Farias; Jair C. Leite**, p. 127-166, 2006.

ZHANG, L. et al. Efficient conflict driven learning in a boolean satisfiability solver.
In: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design.
IEEE Press. p. 279-285. 2001.