UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PAULO ALEXANDRE DA SILVA COSTA

UMA FERRAMENTA PARA ANÁLISE AUTOMÁTICA DE MODELOS DE CARACTERÍSTICAS DE LINHAS DE PRODUTOS DE SOFTWARE SENSÍVEL AO CONTEXTO

FORTALEZA, CEARÁ

2012

PAULO ALEXANDRE DA SILVA COSTA

UMA FERRAMENTA PARA ANÁLISE AUTOMÁTICA DE MODELOS DE CARACTERÍSTICAS DE LINHAS DE PRODUTOS DE SOFTWARE SENSÍVEL AO CONTEXTO

Dissertação submetida ao programa de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Profa. Dra. Rossana Maria de Castro Andrade

Co-Orientador: Profa. Dra. Fabiana Gomes Marinho

FORTALEZA, CEARÁ

2012

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca de Ciências e Tecnologia

C875f Costa, Paulo Alexandre da Silva.
Uma ferramenta para análise automática de modelos de características de linhas de produtos de software sensível ao contexto. / Paulo Alexandre da Silva Costa. - 2013.
121f. : il. , color., ; 30 cm.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Departamento de Computação, Programa de Pós Graduação em Ciência da Computação, Fortaleza, 2013.
Área de Concentração: Engenharia de Software.
Orientação: Profa. Dra. Rossana Maria de Castro Andrade.

1. Software de aplicação - Desenvolvimento. 2. Programas de computador - Verificação. 3. Software - Modelos. I. Título.

CDD 005

PAULO ALEXANDRE DA SILVA COSTA

UMA FERRAMENTA PARA ANÁLISE AUTOMÁTICA DE MODELOS DE CARACTERÍSTICAS DE LINHAS DE PRODUTOS DE SOFTWARE SENSÍVEL AO CONTEXTO

Dissertação submetida ao programa de Pós-Graduação em Ciência da Computação, da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação. Área de concentração: Engenharia de Software Área de concentração: Ciência da Computação

Aprovada em: __/__/____

BANCA EXAMINADORA

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

Dra. Fabiana Gomes Marinho
Universidade Federal do Ceará - UFC
Co-Orientadora

Prof. Dr. Vinicius Cardoso Garcia
Universidade Federal de Pernambuco - UFPE

Profa. Dra.Vânia Maria Ponte Vidal
Universidade Federal do Ceará - UFC

Aos meus Pais.

AGRADECIMENTOS

Agradeço a Deus pois Ele meu a vida e sempre foi Fiel. Deus seja louvado! Agradeço também a minha esposa pelo suporte e força que fizeram ser possível terminar o mestrado. Dedico agradecimentos especiais à minha orientadora, Professora Rossana, pelo norte e confiança oferecidos.

“Sede unânimes entre vós; não ambicioneis coisas altas, mas acomodai-vos às humildes; não sejais sábios em vós mesmos; Romanos 12:16”

(Bíblia Sagrada)

RESUMO

As Linhas de produtos de software são uma forma de maximizar o reuso de software, dado que proveem a customização de software em massa. Recentemente, Linhas de produtos de software (LPSs) têm sido usadas para oferecer suporte ao desenvolvimento de aplicações sensíveis ao contexto nas quais adaptabilidade em tempo de execução é um requisito importante. Neste caso, as LPSs são denominadas Linhas de produtos de software sensíveis ao contexto (LPSSCs). O sucesso de uma LPSSC depende, portanto, da modelagem de suas características e do contexto que lhe é relevante. Neste trabalho, essa modelagem é feita usando o diagrama de características e o diagrama de contexto. Entretanto, um processo manual para construção e configuração desses modelos pode facilitar a inclusão de diversos erros, tais como duplicação de características, ciclos, características mortas e falsos opcionais sendo, portanto, necessário o uso de técnicas de verificação de consistência. A verificação de consistência neste domínio de aplicações assume um papel importante, pois as aplicações usam contexto tanto para prover serviços como para auto-adaptação caso seja necessário. Neste sentido, as adaptações disparadas por mudanças de contexto podem levar a aplicação a um estado indesejado. Além disso, a descoberta de que algumas adaptações podem levar a estados indesejados só pode ser atestada durante a execução pois o erro é condicionado à configuração atual do produto. Ao considerar que tais aplicações estão sujeitas a um grande volume de mudanças contextuais, a verificação manual torna-se impraticável. Logo, é interessante que seja possível realizar a verificação da consistência de forma automatizada de maneira que uma entidade computacional possa realizar essas operações. Dado o pouco suporte automatizado oferecido a esses processos, o objetivo deste trabalho é propor a automatização completa desses processos com uma ferramenta, chamada FixTure (FixTure), para realizar a verificação da construção dos modelos de características para LPSSC e da configuração de produtos a partir desses modelos. A ferramenta FixTure também provê uma simulação de situações de contexto no ciclo de vida de uma aplicação de uma LPSSC, com o objetivo de identificar inconsistências que ocorreriam em tempo de execução.

Palavras-chave: Linha de produtos de software. Sensibilidade ao Contexto. Modelos de características. Ferramentas automáticas de Verificação. Software Sensível ao Contexto. Adaptação de software.

ABSTRACT

Software product lines are a way to maximize software reuse once it provides mass software customization. Software product lines (SPLs) have been also used to support context-aware application's development where adaptability at runtime is an important issue. In this case, SPLs are known as Context-aware software product lines. Context-aware software product line (CASPL) success depends on the modelling of their features and relevant context. However, a manual process to build and configure these models can add several errors such as replicated features, loops, and dead and false optional features. Because of this, there is a need of techniques to verify the model consistency. In the context-aware application domain, the consistency verification plays an important role, since application in this domain use context to both provide services and self-adaptation, when it is needed. In this sense, context-triggered adaptations may lead the application to undesired state. Moreover, in some cases, the statement that a contex-triggered adaptation is undesired only can be made at runtime, because the error is conditioned to the current product configuration. Additionally, applications in this domain are submitted to large volumes of contextual changes, which imply that manual verification is virtually not viable. So, it is interesting to do consistency verification in a automated way such that a computational entity may execute these operations. As there is few automated support for these proccesses, the objective of this work is to propose the complete automation of these proccesses with a software tool, called FixTure, that does consistency verification of feature diagrams during their development and product configuration. FixTure tool also supports contextual changes simulation during the lifecycle of a CASPL application in order to identify inconsistencies that can happen at runtime.

Keywords: Software product lines. Context-awareness. Feature models. Automatic verification tools. Context-aware software. Software Adaptation.

LISTA DE FIGURAS

Figura 2.1 Comparação entre as estratégias leve e pesada em LPS e o desenvolvimento de um único sistema (Adaptado de (MCGREGOR et al., 2002)) 22

Figura 2.2 Comparação entre os custos para desenvolver *"n"* tipos de sistemas como sistemas isolados um do outro e o custo do equivalente em uma abordagem para LPS (Adaptado de (POHL; BöCKLE; LINDEN, 2005)) 23

Figura 2.3 Tríade das atividades essenciais (Adaptado de (NORTHROP, 2002)) 26

Figura 2.4 Engenharia de domínio: entradas e saídas (Adaptado de (NORTHROP, 2002)) 26

Figura 2.5 Engenharia de aplicação: entradas e saídas (Adaptado de (NORTHROP, 2002)) 26

Figura 2.6 Exemplo de uma diagrama de características 28

Figura 2.7 Escopo horizontal e vertical de software reutilizáveis (Adaptado de (CZARNECKI; EISENECKER, 2000)) 29

Figura 2.8 Exemplo de regra de dependência (Adaptado de (CZARNECKI; EISENECKER, 2000)) 30

Figura 2.9 Dois produtos que compartilham módulos (Adaptado de (BACHMANN; BASS, 2001)) 30

Figura 2.10 Descrição dos dois produtos com pontos de variação (Adaptado de (BACHMANN; BASS, 2001)) 30

Figura 2.11 Alternativas para a variante (Adaptado de (BACHMANN; BASS, 2001)) ... 31

Figura 4.1 Composição dos modelos de uma LPSSC 42

Figura 4.2 Meta-modelo do diagrama de Características: representação gráfica 43

Figura 4.3 Meta-modelo do diagrama de Características: representação textual 44

Figura 4.4 Meta modelo das regras de composição: representação gráfica 45

Figura 4.5 Meta modelo das regras de composição: representação textual 46

Figura 4.6 Exemplo de modelo de sistema 47

Figura 4.7 Um produto está meta descrito no diagrama de características 49

Figura 4.8 Um produto que está desobedecendo a regra de composição da Figura 4.6 .. 50

Figura 4.9 Meta-modelo do diagrama de contexto: representação gráfica 51

Figura 4.10 Meta-modelo do diagrama de contexto: representação textual 52

Figura 4.11 Instância do meta-modelo do diagrama de contexto 52

Figura 4.12 Evolução do *snapshot* 53

Figura 4.13 Representação textual das regras de contexto: parte 1 53

Figura 4.14 Representação textual das regras de contexto: parte 2 54

Figura 4.15 Representação gráfica das regras de contexto 55

Figura 4.16 Instância do meta-modelo das regras de contexto 55

Figura 5.1 Diagramas de características e de contexto do exemplo 58

Figura 5.2 Regras de contexto do exemplo 59

Figura 5.3 Atributos referenciados: $m_1.attr_1$ e $m_2.attr_1$ 61

Figura 5.4 Uma possível configuração 61

Figura 5.5 Fórmulas referenciando o mesmo atributo 62

Figura 5.6 Cobertura dos valores para as fórmulas atômicas criadas para um mesmo atributo 66

Figura 5.7 Fluxograma do PRECISE (MARINHO, 2012) 70

Figura 5.8 Exemplo de regra em EVL 71

Figura 5.9 Regra em EVL para verificação do diagrama de contexto 71

Figura 5.10 Regras de boa formação sobre as regras de composição e de adaptação 72

Figura 5.11 Especificação em EVL para a quarta fase 78

Figura 5.12 Simples diagrama de características 83

Figura 5.13 Codificação de uma LPS 84

Figura 5.14 Diagrama de características para a simulação da Figura 5.15 87

Figura 5.15 Comportamento em profundidade 87

Figura 5.16 Comportamento em largura 88

Figura 5.17 Visão inicial da ferramenta FixTure 89

Figura 5.18 Validação inicial 89

Figura 5.19 Iniciando o Precise 90

Figura 5.20 Resultado da Fase 1 90

Figura 5.21 Erro na Fase 1 91

Figura 5.22 Erro na Fase 2: Regras de contexto inconsistentes entre si 91

Figura 5.23 Erro na Fase 2: Regras de contexto que tornam qualquer configuração de produto desobediente a alguma regra de composição 91

Figura 5.24 Erro na Fase 3: Detecção de anomalias do tipo *"falso opcional"* (video e texto são características falso opcionais) 91

Figura 5.25 Erro na Fase 3: Detecção de anomalias do tipo *característica morta opcional"* 92

Figura 5.26 Fase 4: Produto a ser verificado em relação à obediência às regras de composição 92

Figura 5.27 Resultado da simulação 93

Figura 6.1 Abordagem empregada no MobiLine (MARINHO et al., 2012) 96

Figura 6.2 Diagrama de contexto do "Guia de Visita Móveis 99

Figura 6.3 Diagrama de características do MobiLine (parte 1) 101

Figura 6.4 Diagrama de características do MobiLine (parte 2) 102

Figura 6.5 Diagrama de características do MobiLine (parte 3) 102

Figura 6.6 Simulação do produto *"littleGreatTour"* da *"MobiLine"* usando o comportamento 1 107

Figura 6.7 Simulação do produto *"littleGreatTour"* da *"MobiLine"* usando o comportamento 2 108

LISTA DE TABELAS

Tabela 3.1 Comparação dos trabalhos relacionados 39

Tabela 4.1 Produto que desobedece à regra de composição 48

Tabela 5.1 Fórmulas atômicas que referenciam o mesmo atributo $o_1.attr_1$ 63

Tabela 5.2 Possíveis combinações das fórmulas atômicas que referenciam o mesmo atributo $attr_1$ 64

Tabela 5.3 Possíveis combinações das fórmulas atômicas que referenciam o mesmo atributo $attr_1$ 65

Tabela 5.4 Mapeamento de modelo de sistema para lógica proposicional (adaptado de (BENAVIDES; SEGURA; RUIZ-CORTÉS, 2010)) 67

Tabela 5.5 Desobediência condicionada 79

Tabela 6.1 Configuração do littleGreatTour 106

SUMÁRIO

1 INTRODUÇÃO 16
1.1 Contextualização 16
1.2 Motivação 17
1.3 Objetivos 18
1.4 Estrutura do documento 19
2 FUNDAMENTAÇÃO TEÓRICA 20
2.1 Linhas de produtos de software 20
2.2 Sensibilidade ao contexto e adaptação dinâmica de software 32
2.3 Conclusões 35
3 TRABALHOS RELACIONADOS 36
3.1 Ferramentas e abordagens 36
3.2 Comparação entre os trabalhos relacionados 38
3.3 Conclusão 40
4 MODELAGEM 41
4.1 Meta-modelos 41
4.2 Meta-Modelo de uma LPSSC 42
4.2.1 Diagrama de características 43
4.2.2 Regras de Composição 44
4.2.3 Diagrama de Contexto 49
4.2.4 Regras de Contexto 50
4.2.5 Regras de boa formação 55
4.3 Conclusões 57
5 MECANISMO DE VERIFICAÇÃO AUTOMÁTICA 58
5.1 Exemplo 58
5.2 Representação de uma LPSSC como fórmulas proposicionais 59
5.3 Implementação do PRECISE 68
5.3.1 Fase 1 - Verificação de diagramas 69
5.3.2 Fase 2 - Verificação de regras 72

5.3.3 Fase 3 - Verificação de regras 77
5.3.4 Fase 4 - Verificação de produto configurado 78
5.3.5 Fase 5 - Simulação 78
5.3.5.1 Processamento em profundidade 84
5.3.5.2 Abordagem em largura 85
5.4 Interfaces gráficas e uso da FixTure 89
5.5 Conclusões 93
6 AVALIAÇÃO PRELIMINAR 95
6.1 MobiLine 95
6.1.1 Diagrama de característica 96
6.1.2 Regras de composição 97
6.1.3 Diagrama de contexto 98
6.1.4 Regras de contexto 100
6.1.5 Relacionamentos 101
6.2 Metodologia 102
6.3 Problemas identificados pelo PRECISE e Fixture 103
6.4 Simulação 105
6.5 Conclusões 108
7 CONCLUSÃO 110
7.1 Contribuições e resultados alcançados 110
7.2 Trabalhos Futuros 112
REFERÊNCIAS BIBLIOGRÁFICAS 114

1 INTRODUÇÃO

Esta dissertação propõe uma ferramenta, chamada *FixTure*, que propõe a automatização do mecanismo Processo de veRificação para modElos de CaracterístIcas Sensíveis ao ContExto (PRECISE) (MARINHO, 2012) de verificação do modelo de características para linhas de produtos de software sensíveis ao contexto. Além disso, esta ferramenta oferece uma simulação do ciclo de vida de aplicações sensíveis ao contexto oriundas deste tipo de linha.

Neste Capítulo, é apresentada a definição do problema tratado nesta dissertação de mestrado. Na Seção 1.1, é discutida uma contextualização do tema de pesquisa. Na Seção 1.2, é apresentada a motivação para o desenvolvimento deste trabalho. Na Seção 1.3, os objetivos são detalhados. Finalmente, 1.4 a organização do restante deste documento é descrita.

1.1 Contextualização

Uma LPS é uma coleção de sistemas de software, chamada por (FRAKES; ISODA, 1994) de domínio, que compartilham um conjunto gerenciado de características relacionadas a um segmento específico do mercado. Estes mesmos sistemas são construídos de forma pré-determinada a partir de um conjunto chamado na literatura de artefatos núcleo com o objetivo primário de obter vantagens econômicas (NORTHROP, 2002), (POHL; BöCKLE; LINDEN, 2005). Ao prover a customização de software em massa, as LPSs são uma forma de maximizar o reuso de software.

Além de características comuns, os sistemas de software também têm características próprias que os distinguem uns dos outros. Esse aspecto é chamado de variabilidade em LPS e é o responsável por permitir a existência de produtos diversificados. A modelagem da variabilidade consiste em definir a "forma pré-determinada" de se construir produtos da LPS. Uma das formas de se modelar a variabilidade é através de um modelo de características que é uma árvore onde a raiz representa um sistema de software e seus nós descendentes, as características do sistema. Ademais, restrições condicionais podem ser estipuladas sobre o modelo de características a fim de aumentar a expressividade do próprio modelo.

Quanto mais detalhado e aprofundado for o modelo de características, mais produtos são esperados. De uma forma geral, em uma LPS, é esperada uma quantidade mínima de três produtos para que ela seja economicamente vantajosa (MCGREGOR et al., 2002). Na prática, os modelos de características tendem a produzir uma quantidade bem maior de produtos. Consequentemente, a chance de algum produto ser configurado em desacordo com o que foi especificado no modelo de características aumenta. Assim, é desejável buscar uma forma de analisar se o produto está em acordo com a especificação do modelo de característica.

Paralelamente, o interesse em sistemas adaptativos tem crescido e uma variedade de técnicas existentes permite que um software se adapte dinamicamente em tempo de execução (MCKINLEY et al., 2004). Esse crescimento se deve ao amadurecimento da computação ubíqua visionada por Mark Weiser em (WEISER, 1999), a qual concebe a dissolução dos tradi-

cionais limites das interações entre homem e máquina de forma a se tornarem completamente transparentes para o usuário. Além disso, esse crescimento se deve também à computação autonômica (KEPHART; CHESS, 2003) que lida com os desafios e soluções envolvendo a capacidade de uma aplicação realizar auto-gerenciamento (MCKINLEY et al., 2004).

A adaptação de software pode acontecer de duas formas: por parâmetro e por composição (MCKINLEY et al., 2004). O primeiro permite o ajuste de comportamento mediante a variação de um parâmetro específico e este permite a reposição de algoritmos e estruturas do programa obedecendo a interfaces bem definidas.

Na adaptação por composição, é necessário definir quando, onde e como adaptar o software. Do ponto de vista da computação autonômica, é interessante que o próprio software contenha estas três informações. Para saber onde adaptar as aplicações é necessário que uma arquitetura de referência seja concebida de forma a ser seguida pela aplicação qualquer que seja o seu estado. Para saber quando adaptar, a aplicação precisa ser notificada sobre quais informações precisam ser monitoradas a fim de que quando certos valores são atingidos a própria aplicação indique que uma adaptação ocorra. Uma vez que uma aplicação sabe quando e onde adaptar, ela precisa conhecer as suas regras de adaptação que vão definir quais partes (*como*) da aplicação sofrerão mudanças.

Em (CLEMENTS; NORTHROP, 2001a), uma terceira classificação é proposta, chamada extensão, que se diferencia da adaptação por composição pois permite a inserção de novas estruturas e algoritmos sem exigir uma interface fixa.

Conforme (HALLSTEINSEN et al., 2006), (MARINHO et al., 2010) e (PARRA; BLANC; DUCHIEN, 2009) ressaltaram, as LPSs podem ser usadas no desenvolvimento de aplicações sensíveis ao contexto. Quando usadas neste domínio, as LPSs passam a ser chamadas de LPSSCs.

1.2 Motivação

Atualmente, a maioria das abordagens em engenharia de LPS concentra-se em uma configuração estática dos produtos. Esses produtos são gerados resolvendo os pontos de variação da LPS antes da implantação da aplicação. Uma vez configuradas, não é esperado que essas aplicações mudem futuramente. Contudo, em um domínio sensível ao contexto, várias mudanças podem ocorrer como resultado das mudanças de contexto, exigindo que as aplicações sejam adaptáveis.

Sistemas adaptáveis requerem um paradigma de desenvolvimento de software que suporte a checagem automática de suas propriedades funcionais e não funcionais (MCKINLEY et al., 2004). Adicionalmente, é desejável que o próprio software seja capaz de auto configuração. Para tanto, eles precisam saber **quando**, **onde** e **como** se adaptar. No que diz respeito ao ***"quando"***, o software pode estar ciente do contexto que o circunda e utilizar essa informação em benefício da aplicação. Para saber ***"onde"*** se adaptar, é necessário que sejam pré-estabelecidas regras de contexto que especificam novos valores para os atributos e remoções ou adições de características ao produto corrente. No que se refere ao ***"como"***, é necessário oferecer uma

arquitetura de referência que contenha em si todos os possíveis estados que a aplicação possa assumir de forma que ela seja consultada no momento da adaptação.

Uma LPSSC oferece a arquitetura de referência da própria LPS, pois todos os produtos configuráveis estão meta-descritos no modelo de características (o modelo de características descreve todos os possíveis produtos que podem ser derivados do mesmo). Além disso, a arquitetura de referência deve ser enriquecida no sentido de também oferecer a modelagem do contexto de forma que a aplicação saiba o que monitorar e quando adaptar-se.

Uma vez que a LPSSC pode ser uma solução para a adaptabilidade de software sensível ao contexto, os métodos de geração e verificação automática de produtos da LPS tradicional precisam ser estendidos para dar suporte à adaptação contextual e auto-configuração.

Enquanto a LPSSC pode ser uma solução para alcançar os requisitos de adaptação em softwares sensíveis ao contexto, os problemas originais da LPS como checagem dos produtos configurados são exponencializados dado o grande fluxo de mudanças de dados contextuais que é esperado acontecer.

Considerando esse cenário, torna-se essencial verificar se os modelos de características são construídos de forma correta e assegurar que nenhuma das restrições estabelecidas seja violada, uma vez que a corretude e a consistência dos mesmos permitem a derivação de produtos móveis e sensíveis ao contexto corretos e consistentes. Com o objetivo de minimizar defeitos e inconsistências nos modelos de características, (MARINHO, 2012) propôs um processo de verificação baseado em linguagens formais, denominado PRECISE. No entanto, este processo é executado manualmente, o que pode ocasionar a inclusão de erros devido ao mau uso do mesmo pelos engenheiros de software e pode também dificultar a sua adoção. Nesse sentido, este trabalho de pesquisa se propõe a automatizar a abordagem apresentada no PRECISE de modo a prover uma ferramenta para checar a corretude e a consistência dos modelos de características no domínio de aplicações sensíveis ao contexto e, com isso, reduzir significativamente os erros inseridos no uso manual do processo e facilitar o seu uso.

1.3 Objetivos

O objetivo principal deste trabalho é desenvolver a ferramenta *FixTure* para auxiliar o engenheiro de software a projetar e verificar automaticamente os modelos de características e os modelos de contexto de uma LPSSC, bem como verificar os produtos derivados e produtos adaptados a partir desses modelos.

Para atingir o objetivo aqui proposto, foram definidos os seguintes objetivos específicos:

- revisão da literatura focada em modelos para definição da linhas de produtos de softwares e do ambiente sensível ao contexto, abordagens para verificações formais e ferramentas já desenvolvidas,
- extensão do meta-modelo de característica e do meta-modelo de contexto descrito em

(MARINHO; ANDRADE; WERNER, 2011) e (MARINHO, 2012),

- estruturação da execução da verificação automática do modelos referentes a uma Linhas de Produtos de Software Sensíveis ao Contexto,
- desenvolvimento de um método de simulação do ciclo de vida de uma aplicação sensível ao contexto,
- análise de tecnologias disponíveis para implementação da ferramenta,
- implementação da ferramenta *FixTure*.

1.4 Estrutura do documento

Esta dissertação contém, além da introdução, seis capítulos descritos a seguir:

- Capítulo 2 apresenta definições sobre LPS, sensibilidade ao contexto e adaptação dinâmica de software.
- Capítulo 3 aborda os trabalhos relacionados.
- Capítulo 4 detalha os meta-modelos estendidos.
- Capítulo 5 apresenta o método de verificação automático, o mecanismo de simulação do ciclo de vida da aplicação sensível ao contexto e a ferramenta propostos neste trabalho.
- Capítulo 6 oferece uma avaliação preliminar usando o Mobiline (MOBILINE, 2012).
- Capítulo 7 traz as conclusões e considerações finais desta pesquisa, incluindo as contribuições, resultados alcançados e limitações, bem como direcionamentos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste Capítulo é apresentada a fundamentação teórica relacionada a este trabalho de pesquisa. Na Seção 2.1, as definições de LPS e LPSSC são apresentadas. Na Seção 2.2, o conceito de sensibilidade ao contexto é introduzido. Por fim, na Seção 2.3, as considerações finais deste Capítulo são apresentadas.

2.1 Linhas de produtos de software

O termo *"crise do software"* foi primeiramente usada em 1969 para descrever as crescentes sobrecarga e frustração que ocuparam o desenvolvimento e a manutenção de software (GRISS, 1993). Tão grande era a crise do software que foi o tema principal da conferência na qual praticamente nasceu a engenharia de software (KRUEGER, 1992).

Parte dessa frustração se deveu ao fato que os métodos de desenvolvimento de software não acompanharam o crescimento da demanda por aplicações mais complexas, mais variadas, maiores e para servir mais pessoas. Some-se a isso, o fato que os ciclos de desenvolvimento diminuíram de anos para meses. Embora diversas abordagens tenham sido propostas, elas eram *"ad hoc"* de tal forma que a cada novo problema, novas abordagens deveriam ser pensadas. Notou-se à época, que quanto mais código novo era produzido, mais cara e difícil a manutenção de software se tornava (GRISS, 1993).

A solução para a crise do software começava então a aparecer: escrever menos código. Para escrever menos código, era necessário que se usasse código que já estava escrito. Portanto, era necessário reusar.

O reuso de software pode ser descrito como a utilização de software ou conhecimento existente para construir novo software, tendo como intuito melhorar a qualidade e a produtividade no desenvolvimento de software (FRAKES; KANG, 2005). O reuso é também uma das formas mais efetivas de aumentar a consistência do software, encurtar o tempo necessário para chegar ao mercado e reduzir os custos de desenvolvimento e manutenção (GRISS, 1993).

À primeira vista, parece que tudo o que é necessário é coletar componentes já desenvolvidos e testados e incentivar o uso destes. Contudo, o sucesso do desenvolvimento de software depende de mais fatores do que simplesmente o eventual reuso de componentes. Em (TRACZ, 1986), é argumentado que a grande maioria dos motivos que impedem o reuso de software é de caráter, social e econômico. Há ainda fatores logísticos e organizacionais que necessitam ser superados para a implantação do reuso (GRISS, 1993). Em (FAYAD; SCHMIDT, 1997), os autores citam que quando esses fatores não são vencidos, fatores culturais podem surgir como a crença de que *"se não foi feito por nós, então não funciona"* (ou *"not invented here"*).

Por outro lado, em (MORISIO; EZRAN; TULLY, 2002), ao tentar elencar os principais fatores que afetam o reuso de software, os autores justificam, através de (FRAKES; FOX,

1995), o nível de reuso da empresa não é afetado por linguagens de programação, ferramentas *"CASE"* ou mesmo pela síndrome supracitada, mas sim pela educação em reuso, tipo do nicho de atuação da empresa e disponibilidade de artefatos reusáveis de alta qualidade.

Programadores tem reusado código, sub-rotinas e algoritmos desde que a programação foi inventada. Eles também sabem como adaptar e fazer engenharia reversa (PRIETO-DIAZ, 1993).

A ideia de reuso sistemático foi primeiramente proposta em (MCILROY et al., 1969) e consiste do desenvolvimento planejado e amplo uso de componentes já construídos. De modo semelhante, os trabalhos de (HABERMANN; FLON; COOPRIDER, 1976) e (NEIGHBORS, 1984) aumentam as contribuições na área de famílias de software.

Em (GRISS, 1993), é defendido que a chave para o sucesso no desenvolvimento de software é propor uma abordagem de reuso sistemática, isto é, planejada e não eventual. A aplicação sistemática de reuso de software para prototipagem, desenvolvimento e manutenção mostrou-se uma das maneiras efetivas de melhorar o processo de software, bem como encurtar o tempo de chegada ao mercado. O reuso sistemático é essencial para aumentar a produtividade e qualidade do software ao romper o custo do ciclo da *"redescoberta, reinvenção e revalidação"* de artefatos comuns de software (SCHMIDT; BUSCHMANN, 2003). Pode-se considerar que o passo seguinte no reuso sistemático ocorreu com o desenvolvimento de processos de reuso de software (ALMEIDA et al., 2005).

Aliado à necessidade de métodos mais eficientes para construção de sistemas, o interesse dos desenvolvedores em construir múltiplos sistemas de softwares a partir de requisitos similares também cresceu (MCGREGOR et al., 2002) desde a primeira vez que foi reportado em (PARNAS, 1976). Desde então, houve uma evolução na unidade de reuso que evoluiu de tamanho e complexidade: inicialmente com funções, rotinas e módulos e classes (artefatos leves) e posteriormente com componentes e *"frameworks"* (artefatos pesados). Além disso, ocorreram significativos avanços em linguagens de programação e metodologias de desenvolvimento de software. Embora, isso pareça suficiente para alavancar o nível de reuso de uma empresa e, assim, diminuir gastos, desenvolver aplicações com qualidade dentro do tempo e orçamento planejados permanece difícil (SCHMIDT; BUSCHMANN, 2003).

Pode-se afirmar que essa dificuldade resulta persistente pois a incorporação de um processo de reuso requer que fatores não técnicos sejam satisfeitos (FAYAD; SCHMIDT, 1997). Tal afirmativa pode ser baseada em (MORISIO; EZRAN; TULLY, 2002) onde são citados três fatores-chave para o sucesso de uma iniciativa de reuso que podem ser sintetizados na seguinte sentença: *"comprometimento da alta gerência em considerar atividades essenciais como processos de reuso, modificar os processos já existentes mas que não endossam o reuso, e fatores humanos. Embora, essas atividades pareçam ser padrão para uma iniciativa de reuso, como realizá-las não são"*.

Mesmo com o aumento da complexidade do artefato reusado, os benefícios esperados com o reuso de software não estavam sendo alcançados devido ao fato destes artefatos implicarem o uso de uma abordagem de desenvolvimento de sistemas de baixo para cima (composição de componentes para construir um sistema, ou *bottom up*) (LAGUNA et al., 2003).

Com o reuso sistemático, houve uma mudança de paradigma a partir da qual famílias de software relacionados são construídas em vez de um único sistema de software (FRAKES; ISODA, 1994). Como dito anteriormente, essa mudança de paradigma havia sido concebida em (PARNAS, 1976).

Desta forma, um novo patamar no reuso de software foi concebido, onde arquiteturas, sistemas ou grandes partes destes eram reutilizados (implicando uma abordagem *"top-down"*) para a construção de outros sistemas. Dado esse cenário de evolução do reuso sistemático, das famílias de programas e de abordagens *top-down*, é possível introduzir a definição de Linha de Produtos de Software LPS.

Uma LPS é um conjunto de sistemas de software que pertencem a um determinado nicho do mercado e que compartilham um conjunto gerenciado de características. Além disso, estes sistemas de software são construídos a partir deste conjunto compartilhado de características, denominado de artefatos núcleo. Artefatos núcleo geralmente incluem (mas não somente) a arquitetura, componentes reusáveis, documentação, planos de teste e modelos de domínio. Dentre eles, a arquitetura é o elemento chave (NORTHROP, 2002). Além das semelhanças, produtos em LPS também apresentam diferenças entre si (MCGREGOR et al., 2002).

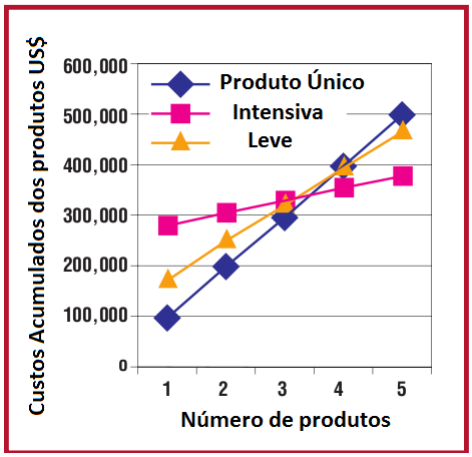


Figura 2.1: Comparação entre as estratégias leve e pesada em LPS e o desenvolvimento de um único sistema (Adaptado de (MCGREGOR et al., 2002))

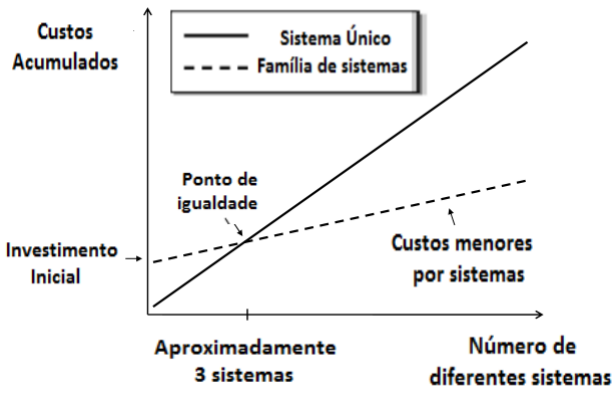


Figura 2.2: Comparação entre os custos para desenvolver *"n"* tipos de sistemas como sistemas isolados um do outro e o custo do equivalente em uma abordagem para LPS (Adaptado de (POHL; BöCKLE; LINDEN, 2005))

(MCGREGOR et al., 2002) e (POHL; BöCKLE; LINDEN, 2005) citam que são necessários pelo menos três produtos para que o desenvolvimento de software através de LPS seja mais vantajoso que o desenvolvimento tradicional de sistemas. As Figuras 2.1 e 2.2 oferecem o gráfico comparativo. Tal afirmação é debatível pois existem inúmeros fatores a serem considerados no cálculo do custo e preço de um software. Um dos principais fatores é o valor percebido pelo cliente que é o valor que o produto representa para o cliente. Adicionalmente, há métricas que auxiliam no cálculo do custo e preço do produto como quantidade de linhas de código e pontos de função. Enquanto que esta oferece valor mais preciso, ela é mais difícil de ser coletado. Em contrapartida, estimar o custo e o preço do software pelas linhas de códigos pode oferecer resultados enganosos embora seja mais fácil. Diante desses fatos, determinar que são necessários pelo menos três produtos para o desenvolvimento através de LPS seja mais lucrativa pode ser enganosa. Em verdade, é preferível que esta quantidade seja estimada pela empresa que deseja implantar a LPS e então, avaliar se será proveitoso a adoção desta metodologia.

Pode-se citar como fatores motivantes da LPS a redução dos custos de desenvolvimento e manutenção, aumento da qualidade, e redução do tempo de chegada ao mercado (POHL; BöCKLE; LINDEN, 2005). A combinação sistemática de customização em massa e uma plataforma comum, chamada em (POHL; BöCKLE; LINDEN, 2005) de engenharia de LPS, permite reusar uma base compartilhada de tecnologia e, ao mesmo tempo, obter produtos próximos às necessidades do cliente. Uma plataforma de software é um conjunto de sub-sistemas de softwares e interfaces que formam uma estrutura comum a partir das quais um conjunto de produtos derivados pode ser eficientemente desenvolvido e produzido (MEYER; LEHNERD, 1997).

Dentre as diversas formas de se implementar uma LPS, três ideias são comuns (MCGREGOR et al., 2002):

- explorar semelhanças entre produtos para **proativamente** reusar os artefatos de software: LPSs usam abordagens proativas que planejam o uso de artefatos em diversos softwares em vez de abordagens *"ad hoc"*;

- encorajar o desenvolvimento centrado em arquitetura: a arquitetura de software é a chave do sucesso para uma LPS. Uma arquitetura que especifica todos os produtos também registra as regiões de semelhança bem como as de diferença; e
- usar uma organização estrutural de duas camadas: uma voltada para criação dos artefatos reusáveis e outra voltada para criação com os artefatos reusáveis.

Em (KRUEGER, 2002), o autor lista as formas de implementar uma LPS de três formas: proativa, reativa e extrativa. A primeira pode ser comparada com o desenvolvimento tradicional em cascata: a análise, arquitetura, design e implementação de todos os produtos ocorrem sequencialmente sem paralelização. Da mesma forma que o cascata, essa abordagem exige que a empresa conheça bem os requisitos de forma que ela possa prever com segurança todas as possíveis variações nos produtos da LPS.

A abordagem reativa estabelece que a análise, arquitetura, design e implementação ocorre para cada produto que possa fazer parte da LPS. Fazendo uma comparação com o desenvolvimento em espiral, em cada ciclo da espiral ocorreria a execução das fases citadas anteriormente. Em (CLEMENTS, 2002), o autor afirma que os artefatos são desenvolvidos à medida que eles são identificados, lembrando, de certa forma, uma abordagem ágil para LPS. Essa abordagem é encorajada quando a empresa não conhece todos os requisitos de todos os possíveis produtos da LPS.

Em terceiro lugar, a abordagem reativa se baseia em reusar softwares existentes como ponto de partida para uma LPS. Partes desses softwares já existentes servirão como artefatos reusáveis para a LPS que se deseja implantar. Essa abordagem é indicada quando a empresa deseja fazer uma transição rápida do desenvolvimento tradicional para o paradigma da LPS.

Naturalmente, as aplicações são classificadas de acordo com o nicho de mercado relacionado ou com as atividades que elas podem desenvolver. Similarmente, podem-se classificar partes do sistema de acordo com a sua funcionalidade. Em (CZARNECKI; EISENECKER, 2000), essas classes de sistema ou de partes de sistemas é denominada como domínio.

Alternativamente, o domínio é definido de outra forma, como *"Um conjunto de aplicações desenvolvidas ou a serem desenvolvidas que compartilham funcionalidades e dados (KANG et al., 1998)"* ou como *"Domínios são famílias de sistemas similares (BAILIN, 1992)"*.

Em (CZARNECKI; EISENECKER, 2000) é oferecida uma definição mais abrangente: *"Domínio é uma área do conhecimento:"*

- com escopo voltado a maximizar a satisfação dos requisitos dos clientes;
- que inclui um conjunto de conceitos e termos compreendidos pelos profissionais da área; e
- que inclui conhecimento de como construir sistemas de software do domínio, ou partes deles.

Uma das definições chave do reuso, presente na LPS, é a engenharia de domínio cuja principal ideia é que muitos sistemas construídos não são novos; na realidade, eles são variações de alguns que já existem. Tecnologias para alta produtividade de software através da engenharia de domínio começaram a aparecer no início da década de 80, mas a aplicação industrial destas tomou forma e volume somente no início do século XXI (FRAKES; KANG, 2005).

É importante ainda ressaltar que os objetivos chave da engenharia de domínio são (POHL; BöCKLE; LINDEN, 2005):

- definir o que é comum e o que é variável na LPS;
- definir o escopo da LPS; e
- definir e construir artefatos reusáveis para prover a variabilidade desejada.

Agregada à LPS, existe o conceito de engenharia de aplicação, que compreende o processo de construir aplicações reusando artefatos criados na fase de engenharia de domínio, artefatos de domínio, e explorando a variabilidade da LPS (POHL; BöCKLE; LINDEN, 2005).

Para a engenharia de aplicação, os objetivos chave são (POHL; BöCKLE; LINDEN, 2005):

- alcançar o máximo de reuso dos artefatos de domínio ao derivar um produto;
- explorar o que é comum e o que é variável da LPS ao derivar um produto;
- documentar os artefatos da aplicação e relacioná-los aos artefatos de domínio;
- vincular a variabilidade às necessidades do produto derivado; e
- estimar os impactos causados pelas diferenças nos requisitos.

Portanto, a separação em duas engenharias indica a separação de interesses no que diz respeito à variabilidade: para reuso e com reuso, respectivamente (POHL; BöCKLE; LINDEN, 2005).

Outra definição para essas duas engenharias pode ser encontrada em (NORTHROP, 2002): a engenharia de domínio é definida como o desenvolvimento dos artefatos núcleo enquanto que a engenharia de aplicação é definida como as atividades voltadas para o desenvolvimento dos produtos. Estas duas engenharias estão ligadas por uma atividade de gerenciamento da LPS conforme pode ser visto na Figura 2.3. Nas Figuras 2.4 e 2.5, entende-se melhor como ocorre o funcionamento das duas engenharias citadas.

Na Figura 2.4 tem-se as entradas necessárias para o desenvolvimento dos artefatos núcleo. Dentre elas, destacam-se as características comuns e as variáveis representadas na imagem por restrições de produtos (*product constraints*). Como saída, há o escopo da linha de produção (que descreve os produtos que uma LPS produz ou pode produzir), os artefatos

núcleo, e o plano de produção (que descreve como os produtos são produzidos a partir dos artefatos núcleo). Na Figura 2.5, o desenvolvimento de produtos (engenharia de aplicação) toma como entrada as três saídas da engenharia de domínio mais os requisitos dos produtos individuais. O resultado final da engenharia de aplicação é o produto configurado.

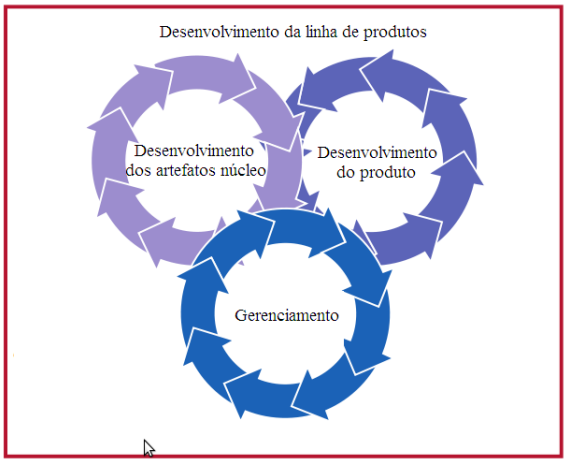


Figura 2.3: Tríade das atividades essenciais (Adaptado de (NORTHROP, 2002))

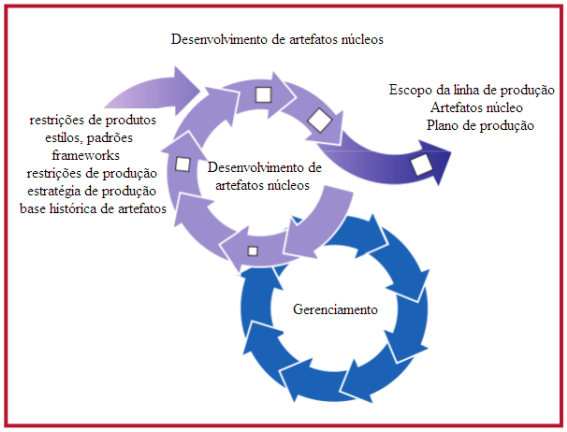


Figura 2.4: Engenharia de domínio: entradas e saídas (Adaptado de (NORTHROP, 2002))

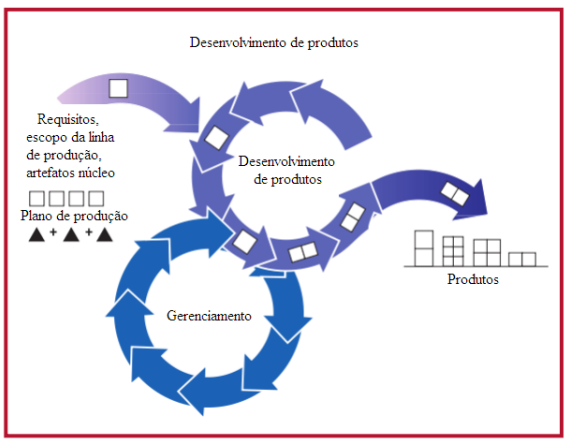


Figura 2.5: Engenharia de aplicação: entradas e saídas (Adaptado de (NORTHROP, 2002))

Para facilitar a customização em massa, os artefatos usados em diferentes produtos têm de ser suficientemente adaptáveis para aderirem a diferentes sistemas produzidos na LPS. Isso significa que através do processo de desenvolvimento é necessário identificar e descrever onde os produtos da LPS podem se diferenciar, em termos das características que eles proveem dos requisitos que eles atendem, ou mesmo dos aspectos da arquitetura subjacente.

Essa flexibilidade é pré-condição para customização em massa. A flexibilidade descrita aqui é chamada de *"variabilidade"* no âmbito da LPS, a qual é a base para customização em massa (POHL; BöCKLE; LINDEN, 2005).

Um importante artefato da fase de engenharia de domínio é o modelo de variabilidade. Este modelo define os pontos de variação e os tipos de variação disponíveis para cada ponto de variação, as dependências, e as restrições de variabilidade. A existência da variabilidade é uma diferença chave entre desenvolvimento de sistemas simples e LPS (CLEMENTS; NORTHROP, 2001b).

A variabilidade de software pode ser definida como: *"Habilidade do sistema ou artefato de software de ser mudado, customizado ou configurado para uso em contexto particular"*(BEUCHE; PAPAJEWSKI; SCHRöDER-PREIKSCHAT, 2004).

A variabilidade em uma LPS é modelada para permitir o desenvolvimento de aplicações customizadas através de reuso pré-definido e artefatos ajustáveis. Portanto, a variabilidade de LPS é o que irá permitir distinguir aplicações de uma mesma LPS (POHL; BöCKLE; LINDEN, 2005).

É consenso utilizar um modelo para representar as diferenças e semelhanças entre vários produtos finais (BEUCHE; PAPAJEWSKI; SCHRöDER-PREIKSCHAT, 2004). Assim, faz-se necessário existir uma maneira de modelar a variabilidade e uma delas é através dos modelos de características (HEIDENREICH, 2009) dado que modelos de características podem ser usados para representar o domínio do problema em termos de semelhanças e diferenças (BEUCHE; PAPAJEWSKI; SCHRöDER-PREIKSCHAT, 2004). Uma documentação adequada da informação sobre a variabilidade deve pelo menos incluir dados suficientes para responder às seguintes questões (POHL; BöCKLE; LINDEN, 2005): o que varia, por que varia e como varia.

Entende-se por característica, aspectos ou qualidades ou funcionalidades de um ou mais sistema de software, que são visíveis para o usuário (KANG et al., 1998). Um diagrama de características é uma árvore com a sua raiz representando um sistema de software e seus nós descendentes, as características. O modelo de características consiste de um ou mais diagramas de características mais informações adicionais como restrições (CZARNECKI; ANTKIEWICZ, 2005).

Na Figura 2.6, tem-se um diagrama de características representando uma Linha de Produtos para Guias de Visita Móveis desenvolvida no projeto Linha de produtos de software para desenvolvimento de aplicações móveis e sensíveis ao contexto (MobiLine) (MARINHO et al., 2010).

De acordo com a definição anterior, as características desse diagrama são: *"Mo-*

bileDevice”, “Set Profile”, “Set Roadmap”, “ContextManagement”, “Persistency”, “Acquisition”, “Access”, “Capture”, “Asynchronous”, “Via external service”, “From sensor” e *“From memory”*.

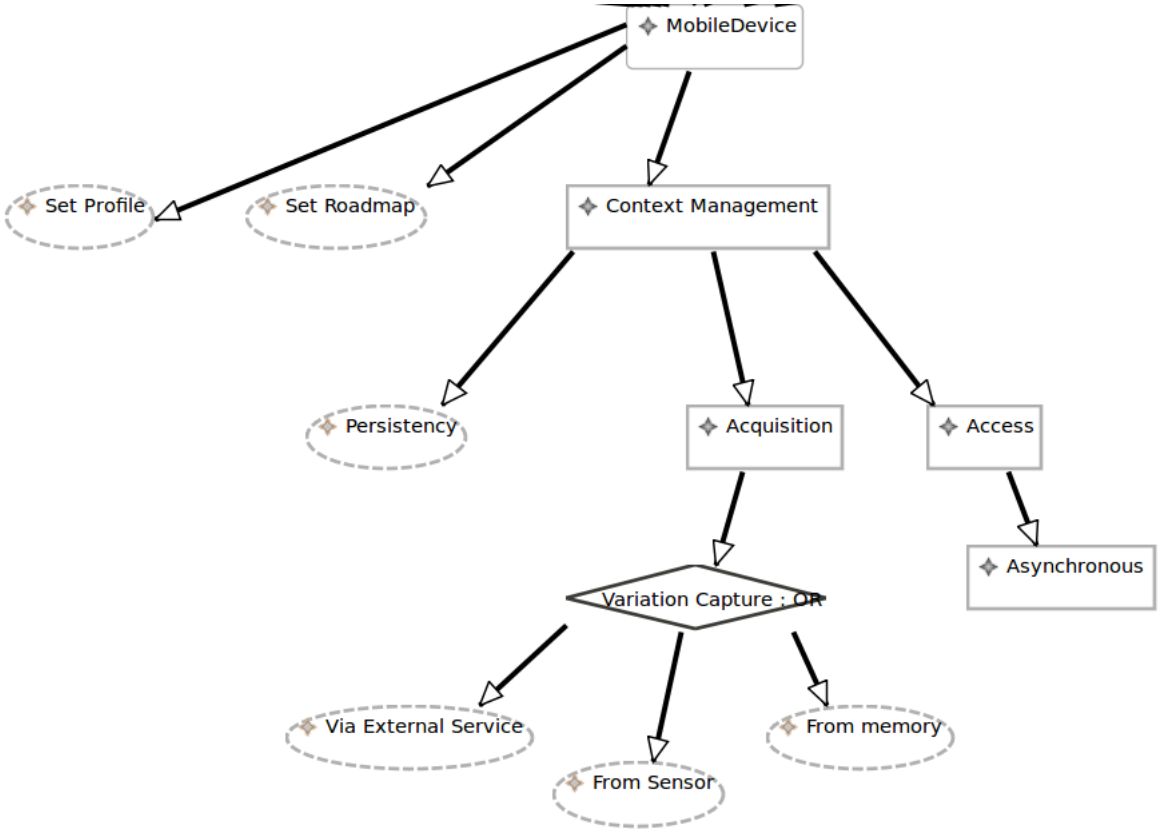
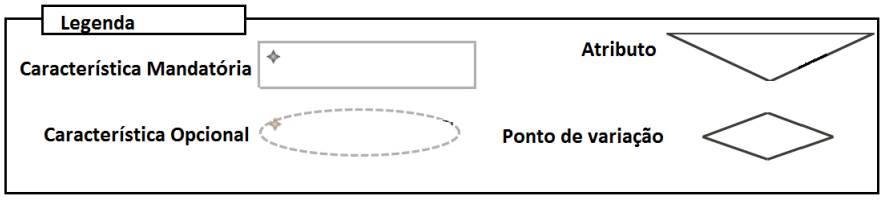


Figura 2.6: Exemplo de uma diagrama de características



Além disso, as características também possuem uma taxonomia definida na literatura (KANG et al., 1998)(CZARNECKI; EISENECKER, 2000): mandatória ou opcional, e alternativa (exclusiva ou não-exclusiva):

1. características mandatórias: todas as aplicações no domínio devem possui-la;
2. característica alternativas (exclusivas ou não): características agrupadas entre si tal que a aplicação pode ter pelo menos uma delas (não-exclusiva) ou exatamente uma (exclusiva); e
3. características opcionais: somente algumas aplicações no domínio as possuem.

Pontos de variação permitem prover implementações alternativas de características

funcionais e não funcionais como diferentes fórmulas de cálculo de taxas ou modos de persistência de dados (CZARNECKI; EISENECKER, 2000). Adicionar mais características opcionais e pontos de variação aumenta o escopo horizontal significando que mais aplicações específicas podem se beneficiar desses elementos (ver Figura 2.7 para explicação de escopo horizontal e vertical).

Na notação adotada na Figura 2.6, os quadrados com bordas cheias representam características mandatórias que são características comuns a todos os possíveis produtos a serem derivados. Os círculos de bordas pontilhadas representam características opcionais. E os losangos representam pontos de variação com a cardinalidade associada (*"OR"* ou *"XOR"*). Assim, alguns potenciais produtos derivados do diagrama de características da Figura 2.6 podem ser identificados como:

- *MobileDevice, SetProfile, ContextManagement, Acquisition, Access, Asynchronous*;
- *MobileDevice, SetProfile, Set Roadmap, ContextManagement, Persistency, Acquisition, Access, Asynchronous*;
- *MobileDevice, SetProfile, ContextManagement, Persistency, Acquisition, Access, Asynchronous, Capture, From Sensor*; e
- *MobileDevice, SetProfile, ContextManagement, Persistency, Acquisition, Access, Asynchronous, Capture, From Memory, Via External Service*

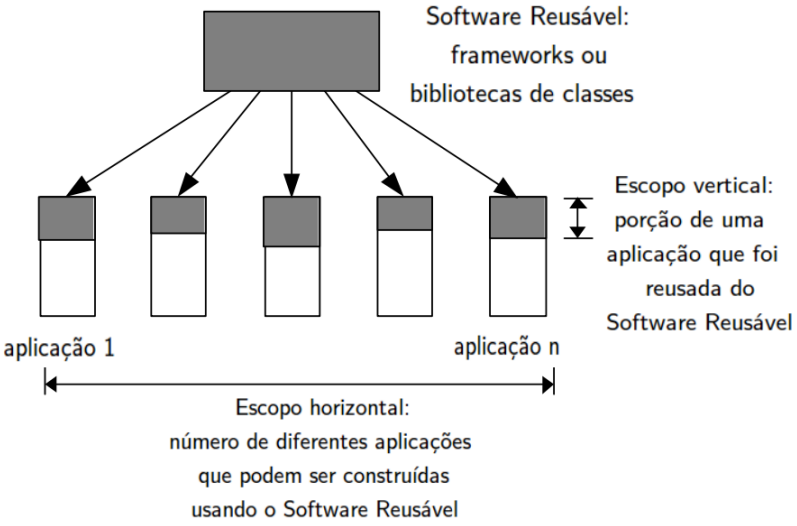


Figura 2.7: Escopo horizontal e vertical de software reutilizáveis (Adaptado de (CZARNECKI; EISENECKER, 2000))

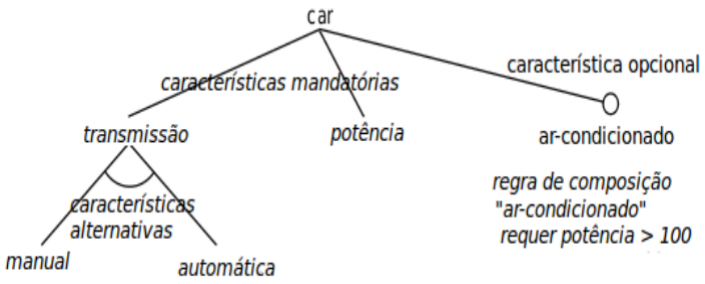


Figura 2.8: Exemplo de regra de dependência (Adaptado de (CZARNECKI; EISENECKER, 2000))

Um ponto de variação é a representação da variabilidade dentro dos artefatos de domínio. Em termos de módulos do sistema, as Figuras 2.9, 2.10 e 2.11 explicam melhor a noção de ponto de variação e variante. Uma vez que é identificada alguma característica comum (representada na Figura 2.9, pela existência dos mesmos módulos A e C bem como as interações entre eles nos dois produtos) ou uma variabilidade (representada na Figura 2.9, pela troca de módulos B e D no mesmo ponto), o sistema pode ser entendido como representado na Figura 2.10 onde *Variante* é exibido na Figura 2.11.

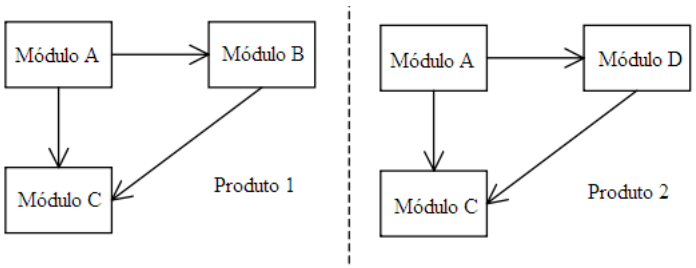


Figura 2.9: Dois produtos que compartilham módulos (Adaptado de (BACHMANN; BASS, 2001))

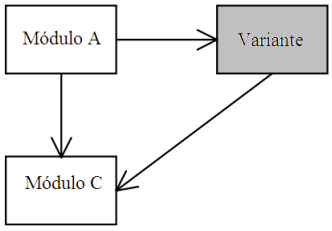


Figura 2.10: Descrição dos dois produtos com pontos de variação (Adaptado de (BACHMANN; BASS, 2001))

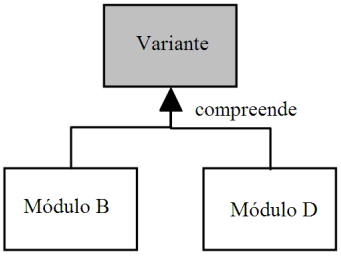


Figura 2.11: Alternativas para a variante (Adaptado de (BACHMANN; BASS, 2001))

A este tipo de variabilidade exemplificado nas Figuras 2.9, 2.10 e 2.11, é possível classificar como sendo do tipo *Ou*. Porém, outros tipos de variabilidade podem existir como, por exemplo, as do tipo *E* ou *XOR* (ou exclusivo), além de dependências e regras específicas para os produtos. Ademais, as características que são comuns e variáveis devem possuir alguma forma de representação, se possível gráfica.

É possível ainda definir dependências entre as características. Em (KANG et al., 1998) são utilizadas dois tipos de regras de composição:

- inclusão: regras de dependência capturam implicações entre as características; por exemplo, *"ar condicionado requer que a potência do motor seja maior que 100 HP"*. Ver Figura 2.8.
- regra de exclusão mútua: essas regras modelam restrições sobre a combinação de características onde a presença de uma característica impede a seleção de outra característica em um mesmo produto e vice-versa.

Portanto, considerando os pontos levantados para modelagem da variabilidade, vários aspectos importantes têm de ser considerados quando desenvolvendo uma ferramenta para o gerenciamento da variabilidade (BEUCHE; PAPAJEWSKI; SCHRöDER-PREIKSCHAT, 2004):

- Os modelos usados para expressar as semelhanças e as diferenças devem ser simples e não devem ser fechados para um domínio específico;
- A variabilidade deve ser possível de instanciar em todos os níveis; e
- A introdução de novas expressões de variabilidade deve ser possível e realizada de forma fácil.

Sendo a LPS um novo paradigma, algumas atividades do desenvolvimento tradicional de software devem ser reformuladas. Assim um outro nome para a atividade de Gerência de Configuração (GC) é gerenciamento de variabilidade que deve prover (THAO; MUNSON; NGUYEN, 2008):

- a seleção de componentes;

- alteração de componentes;
- gerenciamento de componentes modificados;
- manutenção do elo entre os artefatos reusáveis e os produtos derivados; e
- adição, configuração e armazenamento de componentes específicos a um produto.

Outro processo chave em uma LPS é a derivação de produtos (THAO; MUNSON; NGUYEN, 2008). Para se criar produtos válidos a partir de uma LPS, é necessário garantir a boa formação do modelo de entrada (modelo de características) e do modelo de saída (seleção de características) (HEIDENREICH, 2009). Por boa formação, entende-se como a conformidade entre um modelo e um metamodelo subjacente (HEIDENREICH, 2009). No contexto das LPSs, o modelo é representado pelo modelo da família de produtos, e o metamodelo, pelo modelo que contém as restrições sendo estas chamadas de regras de boa formação (HEIDENREICH, 2009).

2.2 Sensibilidade ao contexto e adaptação dinâmica de software

Em (WEISER, 1991), o termo computação ubíqua se refere à disponibilidade integral de dispositivos computacionais no ambiente físico ao mesmo tempo que eles não são mais distinguíveis pelas pessoas de tal modo que eles se tornam invisíveis. Com esta invisibilidade, os dispositivos computacionais devem ser capazes de fazer o que o usuário quer. Assim, os dispositivos funcionais devem ser capazes de se adaptar de acordo com as necessidades do usuário de forma invisível sem intervenção humana ou requisição da atenção do usuário. Desta forma, o contexto é um conceito essencial na construção de sistemas ubíquos, pois é através da análise do contexto que os dispositivos computacionais podem prover serviços relevantes para o usuário. Esses serviços ou aplicações são denominados de sensíveis ao contexto.

Contexto é qualquer informação que pode ser usada para caracterizar a situação de uma entidade. Uma entidade é uma pessoa, lugar ou objeto que é considerado relevante para o usuário, a aplicação ou para a interação entre o usuário e a aplicação (DEY, 2001).

Um sistema sensível ao contexto possui então a capacidade de examinar o ambiente computacional e reagir de acordo com as mudanças nesse ambiente. (SCHILIT; ADAMS; WANT, 1994). Algumas questões surgem: *"como e quando reagir"*, *"onde adaptar"* e *"a adaptação leva a aplicação para um estado não desejado ou não previsto?"*.

Em (SCHILIT; THEIMER, 1994), a computação sensível ao contexto é definida como a habilidade de uma aplicação de um usuário móvel descobrir e reagir às mudanças no ambiente em que se encontra. Desta forma, quanto mais o ambiente muda mais esses sistemas também mudarão. No trabalho citado, é destacado o alto grau de mobilidade do dispositivo que executa este tipo de aplicação e da rapidez de mudança do ambiente ao qual o dispositivo se encontra exposto. Por isso, as informações contextuais do referido trabalho se concentram basicamente em informações de localização.

Contudo, há classificações de contexto mais amplas como em (SCHILIT; ADAMS; WANT, 1994) e em (DEY et al., 1997). Porém, como dito em (ABOWD et al., 1999), estas

classificações são, de certa forma, vagas pois os aspectos tidos como importantes para uma situação podem não ser importantes em outras situações. Assim, é bastante complexo catalogar um conjunto de informações contextuais como relevantes para todas os domínios de aplicações. Contudo, é possível dirigir esforços para determinar o contexto relevante para um determinado domínio de aplicações.

Um sistema passa a ser classificado como sensível ao contexto se ele usa contexto para prover serviços ou informações relevantes para o usuário, de acordo com a tarefa que este realiza (DEY, 2001). Neste trabalho, será considerado que os sistemas sensíveis ao contexto são um subconjunto dos sistemas ubíquos.

Em (SCHILIT; ADAMS; WANT, 1994), há dois termos resgatados posteriormente em (ABOWD et al., 1999). O primeiro deles é a reconfiguração contextual automática que é a capacidade de reconfiguração direcionada pelo contexto em que o dispositivo se encontra. O segundo é a ação disparada por contexto (*context-triggered actions*) que são regras *se... então* onde a condição *se* é avaliada de acordo com o contexto e que, quando satisfeita, pode prover oferta de serviços ou realizar ações.

Ambas as definições ajudam a amadurecer o conceito apresentado em (WEISER, 1999) que afirma que as aplicações ou dispositivos computacionais não devem ser o centro da atenção humana. Esses conceitos estão nitidamente ligados à capacidade do sistema ou dispositivo computacional de se reconfigurar em tempo de execução. Logo, é interessante implementar esses dois conceitos em aplicações sensíveis ao contexto.

Reconfiguração é o processo de adicionar, remover ou alterar componentes. No caso de aplicações sensíveis ao contexto, o aspecto interessante sobre a reconfiguração é como o contexto pode gerar configurações diferentes e quais as adaptações que levam a aplicação de uma configuração a outra (SCHILIT; ADAMS; WANT, 1994).

Assim, alguma forma de oferecer o requisito de reconfiguração deve ser concebida. Na literatura (HALLSTEINSEN et al., 2008), (HALLSTEINSEN et al., 2006), (CETINA; FONS; PELECHANO, 2008), (PARRA; BLANC; DUCHIEN, 2009), é argumentado que o paradigma das LPSs é adequada para oferecer esse requisito. Tal argumentação pode se basear pela arquitetura flexível e pela adaptação que as LPSs oferecem.

Uma segunda forma de classificar aplicações sensíveis ao contexto é agrupá-las em sensibilidade ativa ou passiva. Em sistemas passivos, o contexto é apresentado ao usuário de modo a informá-lo do atual contexto ou de uma mudança no contexto. O sistema é dito passivo, pois não realiza nenhuma adaptação quando ocorre a mudança de contexto. Em um sistema ativo, a mudança de contexto gera a adaptação que é executada pela própria aplicação e não pelo usuário (POSLAD, 2009).

Sistemas sensíveis ao contexto contêm modelos que precisam definir o que o contexto descreve e como contextos são criados, compostos e usados para adaptação (POSLAD, 2009).

No entanto, existem vários problemas relacionados em fazer a adaptação de contexto na prática. Primeiro, as adaptações decorrentes da mudança de contexto podem falhar das

seguintes maneiras: as adaptações podem estar erroneamente especificadas.

Um dos primeiros exemplos de aplicação sensível ao contexto é o *"Active Badge Location System" (WANT et al., 1992)*. Uma vez que a pessoa fosse localizada pela insígnia que ela portava, as chamadas de telefone direcionadas a sua mesa eram direcionadas a mesa da qual a pessoa estivesse mais próxima. Este sistema usa o conhecimento sobre a localização da pessoa para fazer o roteamento das chamadas para a linha fixa telefônica mais próxima. Sensores eram espalhados pelos escritórios, áreas comuns e maiores corredores para detectar as insígnias que as pessoas estavam portando. Obviamente, algumas áreas permaneceram sem sensores para que as pessoas ficassem livre de monitoramento (POSLAD, 2009).

Cada contexto individual em si é definido por um meta-contexto que é a informação que descreve o contexto. Por exemplo, o contexto sobre a localização deve também definir qual o sistema de localização a ser usada bem como as unidades a serem usadas. Novos contextos podem ser criados em tempo real usando sensores situados no ambiente físico como sensores de temperatura ou de pressão (POSLAD, 2009).

Considerando esse cenário, a necessidade de adaptação do comportamento das aplicações em tempo de execução tem exigido o desenvolvimento de aplicações capazes de adaptar suas propriedades e recursos em tempo de execução. As LPSs que fornecem suporte para o desenvolvimento de aplicações sensíveis ao contexto são denominadas **LPSSCs** (FERNANDES; WERNER, 2008).

Nesse contexto, também surgiram as *Linhas de Produtos de Software Dinâmicas*. Segundo (HALLSTEINSEN et al., 2008), uma Linha de produtos de software dinâmica (LPSD) é formada por softwares que são capazes de se adaptar em virtude alterações das necessidades do usuário ou de recursos disponíveis.

No entanto, um software dinâmico não precisa ser sensível ao contexto. Porém, na prática, softwares sensíveis ao contexto usam as informações de contexto para prover serviços aos usuários como a adaptação.

Neste sentido, alguma confusão pode surgir em relação aos termos LPSSC e LPSD. Estipular essa diferenciação é importante, pois o termo dinâmico oferece uma definição abstrata de adaptação, onde o dispositivo pode se adaptar de qualquer forma, inclusive adicionando na LPS pontos de variação em tempo de execução (HALLSTEINSEN et al., 2008). Este tipo de adaptação não é tratada no escopo deste trabalho de dissertação.

Outra diferença entre LPSSC e LPSD é que as LPSSCs têm como foco prover serviços de acordo com as necessidades do usuário, analisando o ambiente em que o mesmo se encontra. Em uma LPSD, esta preocupação não é mandatória.

Contudo, neste trabalho, a dinamicidade oferecida pela LPSSC consistirá apenas em permitir a reconfiguração do produto já derivado em tempo de execução. Essa diferenciação é registrada na literatura (OREIZY et al., 1999) como adaptação aberta (sendo possível, incluir novos comportamentos) e adaptação fechada (não sendo possível incluir novos comportamentos).

A mudança na maioria dos sistemas é inevitável: quanto maior o uso dos mesmos, mais vezes implica em mudanças. (OREIZY; MEDVIDOVIC; TAYLOR, 2008).

A reconfiguração dinâmica de produto refere-se a fazer mudanças na configuração de um produto já implantado enquanto o sistema ainda está executando (LEE; KANG, 2006). A reconfiguração dinâmica é desejável, pois elimina o tempo de *"downtime"* das aplicações garantindo que elas estão sempre disponíveis para os usuários e para outros sistemas (OREIZY; MEDVIDOVIC; TAYLOR, 2008).

Embora, haja um grande interesse na literatura relacionado à reconfiguração dinâmica, a maioria dos trabalhos se concentra em tratar um sistema por vez do que vários. Mesmo em trabalhos voltados para LPS, as abordagens lidam com produtos estaticamente configurados. Em contrapartida aos produtos configurados estaticamente, produtos que são reconfiguráveis dinamicamente devem ser capazes de monitorar o ambiente contextual, validar um pedido de reconfiguração (levando em conta recursos disponíveis e impactos das mudanças) e determinar estratégias para garantir a integridade e a disponibilidade do sistema durante a adaptação (LEE; KANG, 2006).

2.3 Conclusões

Neste capítulo, foram apresentados os conceitos de LPS e sensibilidade ao contexto, neste caso com ênfase na adaptação dinâmica de software.

Dado o conteúdo apresentado, pode-se afirmar que sistemas sensíveis ao contexto precisam de um meta-modelo que defina o contexto relevante. Além disso, LPSs podem ser usadas para prover a arquitetura de referência para as aplicações sensíveis ao contexto. Também foi destacado que modelos de características são amplamente aceitos como opção para modelar a variabilidade das LPSs.

No próximo capítulo, são descritos os trabalhos relacionados a este trabalho de pesquisa no intuito de analisar as vantagens e desvantagens de cada um deles.

3 TRABALHOS RELACIONADOS

Neste capítulo, são apresentadas as abordagens encontradas na literatura que abordam em oferecem, em algum nível, propostas para automatização da análise do modelo de características e que tratam a sensibilidade ao contexto. A Seção 3.1 lista todas as abordagens encontradas na literatura. A Seção 3.2 compara estas abordagens com a FixTure e oferece uma gama de desafios abertos. A Seção 3.3 discute as conclusões a cerca das propostas destacadas.

3.1 Ferramentas e abordagens

A XFeature (XFEATURE, 2011) é uma ferramenta para modelagem de características e é disponibilizada em forma de plugin para a plataforma Eclipse (ECLIPSE, 2012). É voltada para o desenvolvimento baseado na abordagem de famílias de produtos, embora tenha como alvo aplicações espaciais. A XFeature oferece basicamente a visualização e edição gráfica do modelo de características tanto a nível de LPS (exibindo as características da família) como a nível de produto configurado.

A ToolDay (LISBOA et al., 2011) é uma ferramenta de análise de domínio que se propõe a realizar tal tarefa de maneira semi-automática. Esta ferramenta apresenta uma arquitetura bem definida e divida em três camadas: interface gráfica, regras de negócio e persistência de dados. A ToolDay foi construída usando a plataforma *RCP* (*Rich Client Platform*) que aliada a sua arquitetura permite a expansão da ferramenta sem causar danos aos componentes já integrados os quais envolvem o planejamento, a modelagem e validação do domínio e derivação do produto. A ToolDay segue um processo bem definido de fluxo de execução. Na fase de modelagem, é permitido criar interdependências entre as características. Contudo, estes relacionamentos envolvem apenas duas características no estilo de exclusão mútua (ou exclusivo) e inclusão (uma característica exige a outra). Em questão de usabilidade, a ToolDay oferece relatórios de erros encontrados na modelagem além de um tutorial interativo demonstrando os primeiros passos no uso da ferramenta. Sobre os tipos de erros que a ToolDay pode acusar estão a redundância, anomalias (falso opcionais e características mortas) e inconsistência interna (que impede a deerivação de qualquer produto corretamente). A ToolDay não trata a sensibilidade ao contexto na análise do domínio. Além disso, a ToolDay não oferece a criação de regras de relacionamentos mais complexas e nem oferece formalismo na validação dos relacionamentos entre as características.

A FeatureIDE (FEATUREIDE, 2011) é uma IDE voltada exclusivamente para o desenvolvimento orientado a características de LPS. O escopo da FeatureIDE compreende as fases de análise e implementação de domínio, análise de requisitos e geração de software. Tal ferramenta oferece um editor de características gráfico e textual, além de um editor de restrições, baseado em fórmulas proposicionais, com verificação de semântica e sintaxe. Contudo, esta ferramenta não trata a sensibilidade ao contexto.

A UbiFex (FERNANDES; WERNER, 2008) é uma abordagem para a modelagem de características de LPSSC e é construída em cima da Odyssey com base em dois módulos:

UbiFex-Notation e UbiFex-Simulation. O módulo Ubifex-Notation é uma extensão à notação de características que foi desenvolvido para acomodar a representação de contexto no diagrama de características. O módulo Ubifex-Simulation é um mecanismo sensível ao contexto para verificação da consistência de uma configuração de produto. A verificação de consistência apresentada nesse trabalho ocorre apenas no nível dos produtos configurados, excluindo a verificação do modelo, e baseia-se em cenários de execução, não oferecendo assim o formalismo que poderia ser obtido se fosse usada uma linguagem de especificação formal.

A Papyrus (PAPYRUS, 2011) é um componente do subprojeto Model Development Tools (MDT) (MDT, 2011), do Eclipse. A Papyrus tem como objetivo prover ao usuário um ambiente amigável para edição de modelos baseados em Eclipse Modeling Framework (EMF), e prover suporte a Unified Modelling Language (UML) (UML, 2011) e linguagens relacionadas. A ferramenta oferece a funcionalidade de aplicar perfis da UML e inserir restrições em Object Constraint Language (OCL). No entanto, a Papyrus não é voltada para a criação e desenvolvimento de LPSs.

A FeaturePlugin (ANTKIEWICZ; CZARNECKI, 2004) é uma ferramenta desenvolvida sob forma de plugin para o Eclipse. Ela suporta modelagem de características baseada em cardinalidade, especialização e configuração baseadas no diagrama de características. Porém, ela não suporta nenhum tipo de verificação como também não considera a sensibilidade ao contexto.

A Requiline (MASSEN; LICHTER, 2004) é uma ferramenta para a engenharia de requisitos criada para gerenciar de forma eficiente as linhas de produtos. Ela suporta a configuração de produtos, modelagem de características (características e relacionamentos), grupo alternativo (Ou e Ou Exclusivo), checagem de consistência e relatórios de métricas. Embora, a Requiline ofereça o suporte de criar relacionamentos entre as características, ela não trabalha com regras de composição. Além disso, também não suporta a sensibilidade ao contexto.

A DARE (FRAKES; PRIETO-;DIAZ; FOX, 1998) é uma ferramenta CASE para a automação da análise de domínio. Em essência, A DARE não trabalha com LPS, mas sim em um nível mais abstrato do que diagramas de características e, portanto, seu foco é um pouco diferente das outras ferramentas listadas neste capítulo. Em (FRAKES; PRIETO-;DIAZ; FOX, 1998). é citado que foram construídos três protótipos executáveis para DARE, mas não é oferecido um link para download e posteriores reuso e avaliação. Pela descrição em (FRAKES; PRIETO-;DIAZ; FOX, 1998), DARE não oferece sensibilidade ao contexto.

A FeatureMapper (FEATUREMAPPER, 2011) é uma abordagem em forma de ferramenta que combina LPS e desenvolvimento orientado a modelos. Ela oferece mapeamento entre as características do modelos de características e as características selecionadas em um produto configurado (HEIDENREICH; KOPCSEK; WENDE, 2008). A FeatureMapper oferece suporte tanto para a criação do modelo de características como para a criação dos produtos derivados. Contudo, não trata da sensibilidade ao contexto, da consistência, do modelo de características e nem da dos produtos derivados.

Captain Feature (FEATURE, 2012) é uma ferramenta para modelagem de características focada na análise de domínio de famílias de sistemas de software. A sua principal

funcionalidade é criação e edição de diagramas de características. Ela não oferece suporte a edição de regras, nem à derivação de produtos (e consequentemente à validação de produtos). A Captain Feature também não lida o fator de sensibilidade ao contexto. Não há documentação disponível explicando o uso do protótipo disponibilizado.

A S.P.L.O.T (S.P.L.O.T, 2012) é uma ferramenta online para criação e edição de LPS. A S.P.L.O.T oferece a especificação de restrições, contudo requer que as restrições estejam no formato 3-CNF, o que exige habilidade por parte do engenheiro de software em especificar as suas restrições em uma notação tão específica. Além disso, o formato no qual as restrições que podem ser especificadas com S.P.L.O.T permitem uma expressividade menor que a FixTure. Além disso, embora a S.P.L.O.T ofereça a quantidade produtos válidos que podem ser gerados, ela não oferece a possibilidade de realizar a configuração de um produto da LPS. A S.P.L.O.T também não trata a sensibilidade ao contexto e consequentemente não aborda simulação ou ciclo de um produto adaptado da LPS. A S.P.L.O.T oferece a verificação da anomalia *"características mortas"*. A S.P.L.O.T representa o diagrama de características como um diretório de pastas para modelar os relacionamentos entre as características. Aliada a essa representação, a S.P.L.O.T também usa relações de cardinalidade para modelar os grupos alternativos *"OR"* e *"XOR"*. Na homepage da S.P.L.O.T , é possível acessar uma repositório de modelos de características que foram gerados de forma online.

Segundo (SPINCZYK; BEUCHE, 2004), *"pure::variants"* é um plugin para Eclipse que cobre todos os passos do processo de desenvolvimento de uma LPS. "pure::variants" oferece configuração de produtos, verificação de consistência e regras de composição. A edição da LPS acontece em um formato similar ao da "FeaturePlugin": as características são dispostas em formato de diretório de pastas. Porém, a configuração de produtos não é suportada e a sensibilidade ao contexto não é considerada nesta abordagem.

A GEARS (KRUEGER, 2007) é uma ferramenta para desenvolvimento de LPSs. O princípio da GEARS é trabalhar baseado em uma tríade formada por artefatos de software reusáveis de software, perfis de produto (funcionando como uma configuração de produto) e o configurador GEARS que monta automaticamente o produto baseado no perfil de produto selecionado. Ademais, GEARS usa o paradigma de três camadas para o desenvolvimento de SPL: gerenciamento de variabilidade e produção automática, desenvolvimento baseado em artefatos reusáveis e evolução dos perfis de produtos. Segundo (KRUEGER, 2007), a GEARS funciona mais como uma framework teórico do que como protótipo operacional.

3.2 Comparação entre os trabalhos relacionados

De uma forma geral, nenhum deles considera a sensibilidade ao contexto e apenas a Ubifex (FERNANDES; WERNER, 2008) trabalha com o conceito de LPSSC. Um quadro comparativo é fornecido na Tabela 3.1 onde foram criadas as seguintes colunas: protótipo operacional (que indica se um protótipo em forma de software é disponibilizado), configuração de produtos (que indica se, em algum nível, há suporte para a configuração de um produto), Edição de regras (que indica se há suporte à criação e edição de regras sejam de contexto ou de com-

Tabela 3.1: Comparação dos trabalhos relacionados

Ferramenta	Protótipo operacional	Configuração de produtos	Edição de regras	Verificação	Sensibilidade ao contexto
XFeature	■	□	■	◧	□
FeatureIDE	■	■	■	■	□
Odyssey e Ubifex	■	■	◧	◧	◧
Papyrus	■	■	□	□	□
FeaturePlugin	■	■	■	□	□
Requiline	■	■	□	■	□
FeatureMapper	■	■	□	□	□
CaptainFeature	■	□	□	□	□
pure::variants	■	□	■	■	□
S.P.L.O.T	■	■	◧	■	□
GEARS	□	■	□	□	□
ToolDay	■	■	◧	◧	□

■ Possui suporte ◧ Possui suporte parcial □ Não possui suporte

posição), verificação (que indica se alguma verificação, informal ou formal, é realizada pela abordagem ou ferramenta em questão) e sensibilidade ao contexto (que indica se há tratamento, em algum nível, do fator de sensibilidade ao contexto).

Em relação ao quesito edição de regras, há a necessidade de oferecer maior liberdade ao criar e editar regras como permitir regras mais extensas (e de tamanho variável) e referência a atributos. Considerando que algumas ferramentas trabalham com regras contendo uma implicação com um único elemento antecedente e um único elemento como consequente ($A \rightarrow B$, onde *"A"* e *"B"* seriam características), a criação e edição de regras mais complexas é certamente um diferencial.

No que se refere ao tratamento para a sensibilidade ao contexto, também é imperativo oferecer uma modelagem do ambiente contextual que contenha os elementos contextuais relevantes ao domínio da aplicação. Consequentemente, é interessante que esses elementos modelados possam ser referenciados nas regras de contexto. Dadas as abordagens que consideram os elementos contextuais, a modelagem desses elementos é essencial para o tratamento de LPSSC.

Além disso, considerando a alta quantidade de produtos esperados e o fato que esses podem ser originados a partir de reconfigurações disparadas por contexto, outro aspecto importante a ser tratado por abordagens que se propõem a trabalhar com LPSSC é a verificação desses produtos. É importante fornecer algum método de verificação que possa ser automatizado de forma que a verificação possa ser feita automaticamente em lugar de uma verificação manual

que é propensa a erros.

Além disso, a verificação na FixTure segue o formalismo apresentado na tese de doutorado a qual este presente trabalho dá suporte (MARINHO, 2012). Ademais, a FixTure oferece um processo de simulação que enriquece o processo de verificação.

3.3 Conclusão

Este capítulo apresentou os trabalhos relacionados de forma a prover um levantamento bibliográfica das abordagens automatizadas a partir do qual foi possível construir um quadro comparativo. Além disso, com esse levantamento bibliográfico foi possível direcionar os esforços da FixTure no sentido de oferecer soluções para preencher as colunas que não foram totalmente compreendidas nas abordagens relacionadas.

De forma geral, apenas a Odyssey com Ubifex considera a sensibilidade ao contexto como parte de seus requisitos. Contudo, a verificação que ela oferece carece de um formalismo mais acentuado.

Ao se tomar os requisitos apresentados, nenhuma ferramenta os preenche simultaneamente. Neste sentido, a ferramenta exposta nesta dissertação de mestrado se apresenta como preenchedoras das cinco colunas apresentadas no quadro comparativo.

No próximo capítulo, são aprofundados os meta-modelos usados na implementação do mecanismo de verificação automática. Ademais, são detalhados a sua construção e funcionamento, seus principais objetivos e contribuições para a área aderentes como LPS, verificação automática, e adaptação dinâmica de software.

4 MODELAGEM

Neste capítulo é apresentado um conjunto de quatro meta-modelos: diagrama de características, diagrama de contexto, regras de composição e regras de contexto. Estes quatro meta-modelos definem as características comuns e variantes dos produtos, os elementos relevantes de contexto bem como as regras de composição e de contexto de uma LPSSCs. Esses meta-modelos são uma evolução dos modelos apresentados em (MARINHO; ANDRADE; WERNER, 2011) e (MARINHO, 2012). A Seção 4.1 destaca a necessidade de representar a LPSSC através de meta-modelos, de tal forma, que eles possam ser computáveis e lista a necessidade de cada um dos quatro meta-modelos. Na seção 4.2, os quatro meta-modelos listados na Seção 4.1 são formalmente apresentados através de representações gráficas e textuais que são complementares uma a outra. Por fim, a seção 4.3 fecha este capítulo fazendo algumas considerações sobre a modelagem apresentada neste capítulo.

4.1 Meta-modelos

O objetivo principal deste trabalho é oferecer uma mecanismo de verificação automática de uma LPSSC. Para isso, é necessário uma estrutura para definir uma LPSSC que possa ser analisada por um computador de modo que as verificações sobre LPSSC possam ser realizadas automaticamente por um computador. Assim, este trabalho propõe um meta-modelo de dados que representa uma LPSSC que é uma evolução do apresentado em (MARINHO; ANDRADE; WERNER, 2011) e (MARINHO, 2012).

Os meta-modelos propostos nesse trabalho têm a intenção de oferecer uma descrição dos modelos de sistemas sensíveis ao contexto que precisam definir o que os possíveis contextos descrevem e como são criados, compostos e usados para adaptação (POSLAD, 2009). Além disso, este trabalho pretende propor modelos sobre os quais são possíveis realizar algum tipo computação.

A LPSSC lida tanto com elementos de uma LPS tradicional quanto elementos de contexto. Assim, uma LPSSC pode ser entendida como uma linha tradicional acrescida de fatores de contexto. Por LPS tradicional, podemos entender a LPS convencional com um diagrama de características e regras de composição que funcionam como delineadores da estrutura das aplicações pertencentes à LPS.

Conforme será mencionado em 4.2, uma LPSSC lida tanto com elementos de uma LPS tradicional quanto com elementos de contexto, os quais são definidos neste trabalho como todos os elementos necessários para descrever os elementos de contexto relevantes bem como os mecanismos de adaptação. O primeiro consiste em declarar quais são os objetos de contexto de interesse da aplicação, enquanto que o segundo define ações específicas sobre o diagrama de características que são executadas quando certos valores dos objetos de contexto atingem determinados valores. Assim, uma LPSSC pode ser entendida como uma linha tradicional acrescida de fatores de contexto. Vale ressaltar que por LPS tradicional, podemos entender a LPS convencional com um diagrama de características e regras de composição que funcionam

como delineadores da estrutura das aplicações pertencentes à LPS.

4.2 Meta-Modelo de uma LPSSC

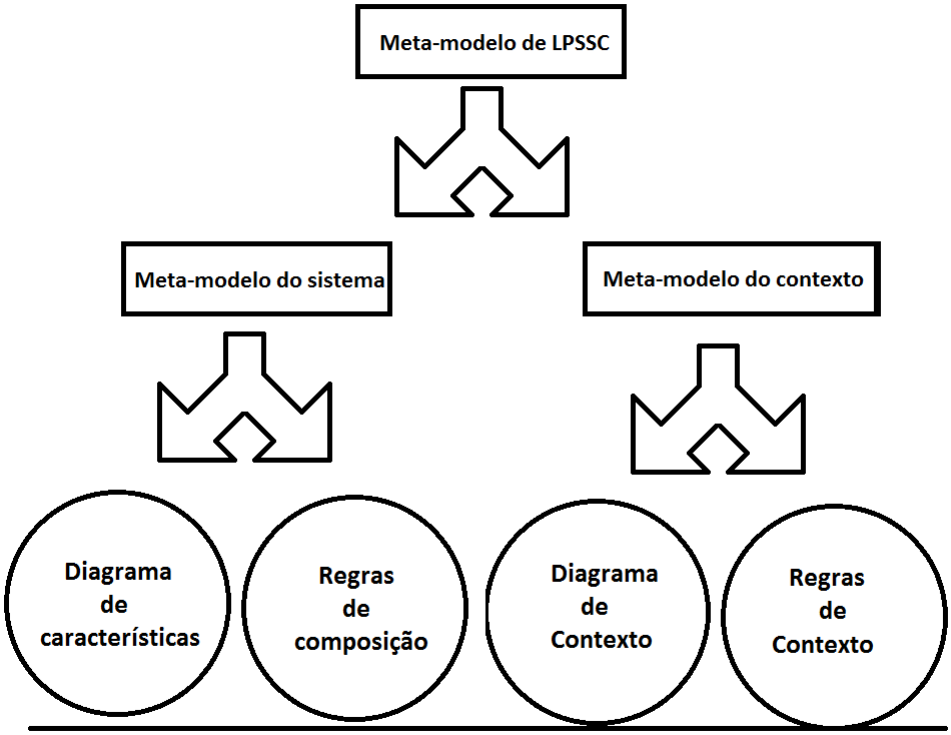


Figura 4.1: Composição dos modelos de uma LPSSC

A Figura 4.1 mostra os elementos do meta-modelo para uma LPSSC. Um meta-modelo de uma LPSSC é composto por dois meta-modelos: o de sistema e o de contexto (MARINHO, 2010) (MARINHO; ANDRADE; WERNER, 2011).

O primeiro meta-modelo lida com as variabilidades e com as semelhanças existentes entre produtos de software (tal qual na LPS tradicional) e o segundo, com os elementos de contexto. Cada um desses sub-modelos é constituído por um diagrama e por regras. No caso do modelo responsável pelos elementos de sistema, (neste trabalho chamado de modelo do sistema) temos o diagrama de características e as regras de composição. Para o modelo responsável pelos elementos de contexto, (neste trabalho determinado de modelo do contexto) temos o diagrama de contexto e as regras de contexto.

Comum aos quatro meta-modelos, há as regras de boa formação que estipulam que tais elementos devem obedecer a certos critérios durante a sua elaboração. Essas regras são detalhadas na seção 4.2.5.

4.2.1 Diagrama de características

O diagrama de características implementado neste trabalho é detalhado na Figura 4.2. Essa notação é própria do Ecore (EMF, 2011). Fazendo um paralelo com a notação da UML, um simples traço unindo duas classes representa uma associação simples. Uma traço com uma seta vazada na extremidade representa herança entre duas classes. Cada classe é representada por um caixa com três compartimentos: o primeiro representa o nome da classes, o segundo, as propriedades, e o terceiro, os métodos da classe. Como o objetivo na Figura 4.2 é mostrar os relacionamentos, os dois últimos compartimentos não são detalhados mas podem ser vistos na representação textual da Figura 4.3.

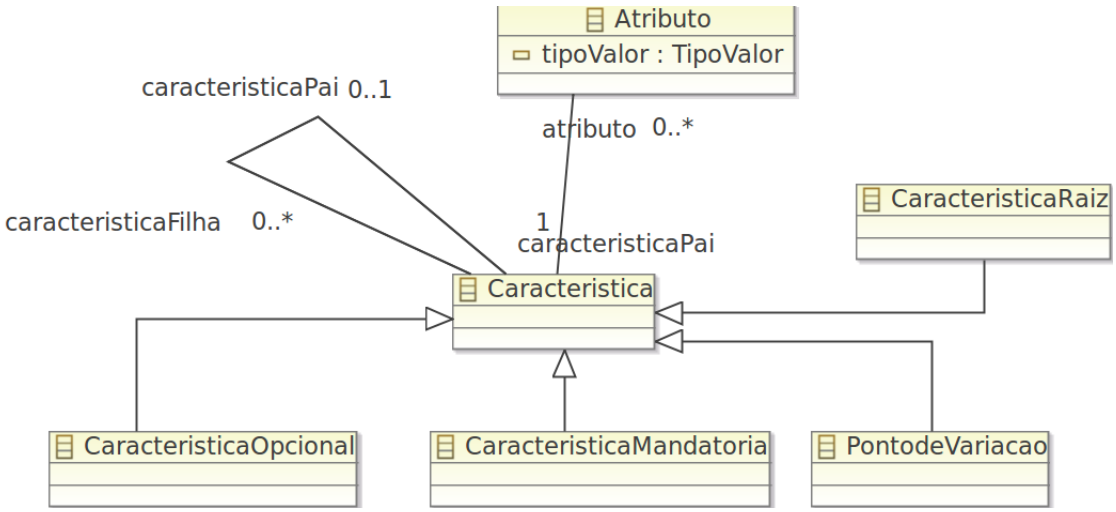


Figura 4.2: Meta-modelo do diagrama de Características: representação gráfica

É fácil perceber que ele atende aos requisitos de cardinalidade do diagrama de características definido em (CZARNECKI; EISENECKER, 2000). Além disso, é importante observar que ele oferece suporte a atributos para as características, algo que pode ser considerado um diferencial do modelo apresentado na tese de doutora (MARINHO, 2012).

Os elementos de sistema compreendem os atributos e as características opcionais, mandatórias, variantes e a raiz. Na modelagem proposta neste trabalho, as características opcionais, mandatórias,variantes e a raiz são modeladas como especialização da super classe *Caracteristica* (*"Caracteristica Opcional"*, *"Caracteristica Mandatoria"*, *"Ponto de Variação"* e *"Caracteristica Raiz"*, respectivamente). A meta-classe *"Caracteristica"* contém um relacionamento de *"zero ou mais"* com *"Atributo"*. A priori, é possível estabelecer um relacionamento entre quaisquer tipos de características (entre mandatória e mandatória ou entre mandatória e opcional, por exemplo). Contudo, alguns desses relacionamentos são proibidos pelas regras de boa formação que são detalhadas mais adiante. Essa mesma especificação gráfica é gerada a partir da classes exibidas na Figura 4.3. A notação apresentada na Figura 4.3 é própria da linguagem Emfatic (Emfatic) (EMFATIC, 2012). A palavra reservada *"ref"* estipula que há um relacionamento enquanto que a palavra reservada *"attr"* especifica um atributo.

Pode-se ver que a estrutura exposta nas Figuras 4.2 e 4.3 virtualmente permite a criação de qualquer diagrama de características pois graças à herança definida entre as classes

e os relacionamentos entre elas é possível estipular vários relacionamentos entre elas limitadas apenas pela memória *RAM* disponível.

A presença de atributos no diagrama de característica aumenta o poder de expressividade do engenheiro de software ao mesmo tempo que insere alguns problemas a serem resolvidos. Na Seção 5.3.2, esses problemas são retomados e tratados.

```
1  class Caracteristica extends Elemento
2  {
3      ref Caracteristica#caracteristicaFilha caracteristicaPai;
4
5      @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid",width="3")
6      ref Caracteristica[*]#caracteristicaPai caracteristicaFilha;
7
8      @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid",width="3")
9      ref Variacao[*]#caracteristicaPai variacoes;
10
11     @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid",width="3")
12     ref Atributo[*]#caracteristicaPai atributo;
13 }
14
15 @gmf.node(label="nome")
16 class CaracteristicaRaiz extends Caracteristica
17 {
18   ref LPS[1]#sistema LpsDoSistema;
19 }
20
21 @gmf.node(label="nome", figure="ellipse", border.style="dash",border.width="2")
22 class CaracteristicaOpcional extends Caracteristica,ElementoCaracteristico{}
23
24 @gmf.node(label="nome",figure="rectangle", border.style="solid",border.width="2")
25 class CaracteristicaMandatoria extends Caracteristica{}
26
27 @gmf.node(label="nome,cardinalidadeMaxima", label.pattern="Variation {0} : {1} ", figure="utils.DiamondFigure")
28 class PontodeVariacao extends Caracteristica,ElementoCaracteristico
29 {
30    attr CardinalidadeMaxima cardinalidadeMaxima;
31 }
```

Figura 4.3: Meta-modelo do diagrama de Características: representação textual

4.2.2 Regras de Composição

Outra parte do meta-modelo de sistema é composta pelas regras de composição. Conforme mencionado no início deste Capítulo, as regras de composição ajudam a delinear a estrutura do diagrama de características. Quando certas condições são atingidas, as regras de composição indicam que certas restrições sobre o modelos de características deverão ser obedecidas pelo aplicação corrente da LPS. Logo, pode-se entender que as regras de composição funcionam como restrições à estrutura do produto derivado.

As regras de composição são descritas na Listagem 4.1 usando Backus–Naur Form

(BNF) (LOUDEN, 2004, Capítulo 3) .

$$
\begin{aligned}
&< regradeComposicao > \rightarrow < operando > implica < operando > \\
&\quad < operando > \rightarrow < operando >< operadorLogico >< operando > \\
&\quad < operando > \rightarrow \text{“atributo”} < operadorRelacional > \text{“valor”} \\
&\quad < operando > \rightarrow < presenca > \text{“caracteristicaOpcional”} \\
&\quad < operando > \rightarrow < presenca > \text{“Variacao”} \\
&\quad < presenca > \rightarrow \text{“presente”} \mid \text{“ausente”} \\
&< operadorLogico > \rightarrow \text{“e”} \mid \text{“ou”} \\
&< operadorRelacional > \rightarrow > | < | \leq | \geq | = | \neq
\end{aligned}
\tag{4.1}
$$

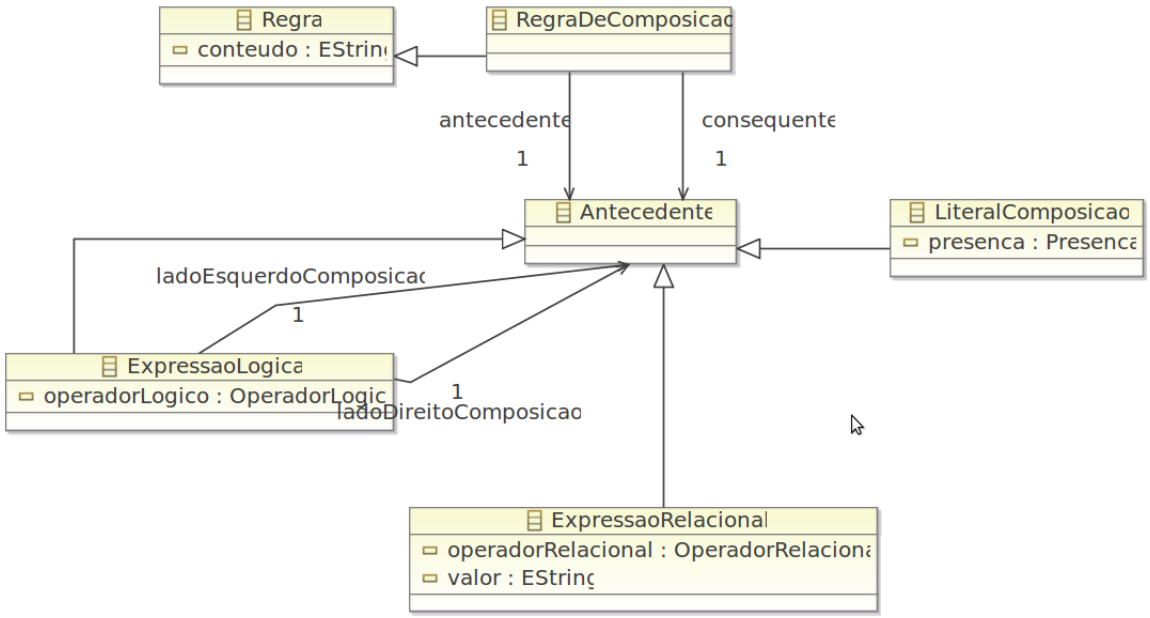


Figura 4.4: Meta modelo das regras de composição: representação gráfica

Para representar esta descrição no meta-modelo deste trabalho, as operações de derivação da BNF são substituídas por herança, conforme podemos ver tanto na representação gráfica (ver Figura 4.4) como na representação textual (ver Figura 4.5). Essa substituição é factível, pois através do mecanismo de herança uma instância da superclasse pode ser substituída por uma da classe especializada, permitindo assim que se construam regras de composição de forma livre porém obedecendo às derivações impostas pelas regras da gramática destacadas acima na Listagem 4.1.

Neste sentido, é possível criar regras com quantidade ilimitada de referências à atributos e características não-mandatórias, aumentando assim a expressividade das regras de composição que podem ser criadas pelo engenheiro de software. Analisando tal meta-modelo, é possível ver que a regra de composição tem dois relacionamentos com a classe *“Antecedente”*: um chamado *“antecedente”* e outro, *“consequente”*. A existência de dois relacionamentos com a mesma classe se deve ao fato que tanto o antecedente como o consequente de uma regra de composição apresentam a mesma estrutura. A classe antecedente é especializado por três

```
class Regra
{
    attr String nome;
    attr String conteudo;
}

@gmf.node(label="nome", label.pattern="Composition Rule {0}")
class RegraDeComposicao extends Regra
{
    @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid", label="antecedent")
    ref Antecedente antecedente;

    @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid", label="consequent")
    ref Antecedente consequente;
}

@gmf.node(label="operadorLogico")
class ExpressaoLogica extends Antecedente
{
    @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid",width="3")
    ref Antecedente ladoDireitoComposicao;

    attr OperadorLogico operadorLogico;

    @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid",width="3")
    ref Antecedente ladoEsquerdoComposicao;
}


@gmf.node(label="nome")
class ExpressaoRelacional extends Antecedente
{

    ref Atributo variaveldaExpressao;

    attr OperadorRelacional operadorRelacional;

    attr String valor;
}

@gmf.node(label="nome")
class LiteralComposicao extends Antecedente
{
   attr Presenca presenca;

   ref ElementoCaracteristico elemento;
}

enum OperadorLogico
{
    AND=0;
    OR=1;
}
```

Figura 4.5: Meta modelo das regras de composição: representação textual

classes: *"ExpressaoRelacional"*, *"ExpressaoLogica"* e *"LiteralComposicao"*. A classe *"ExpressaoRelacional"* estipula uma relacionamento matemático ($>$, $<$, $\geq$, $\leq$, $=$ e $\neq$) entre um atributo e um valor específico. A classe *"ExpressaoLogica"* estabelece dois relacionamentos com a classe *"Antecedente"* unidas por um operador lógico (*"E"* e *"OU"*). Considerando que a classe *"Antecedente"* é especializada por três classes (incluindo a própria classe *"ExpressaoLogica"*), é possível ter um aninhamento de operadores lógicos de tamanho indefinido. Por fim, a classe *"LiteralComposicao"* define se uma característica não-mandatória deve estar presente ou ausente.

Na Figura 4.6 é mostrado um exemplo de uma instância do meta-modelo do sistema: um diagrama de sistema e uma regra de composição. É possível observar que esse exemplo se trata de uma instância do meta-modelo apresentado na Figura 4.2. No caso, é apresentada uma instância de *"CaracteristicaRaiz"* (Mobile Device), quatro de *"CaracteristicaMandatoria"* (ContextManagement, Acquisition, Access e Asynchronous), seis de *"CaracteristicaOpcional"* (Set Profile, Set Roadmap, Persistency, Via External Service, From Sensor e From memory) e uma de *"PontoDeVariacao"* (OR:Capture). Os relacionamentos entre estas

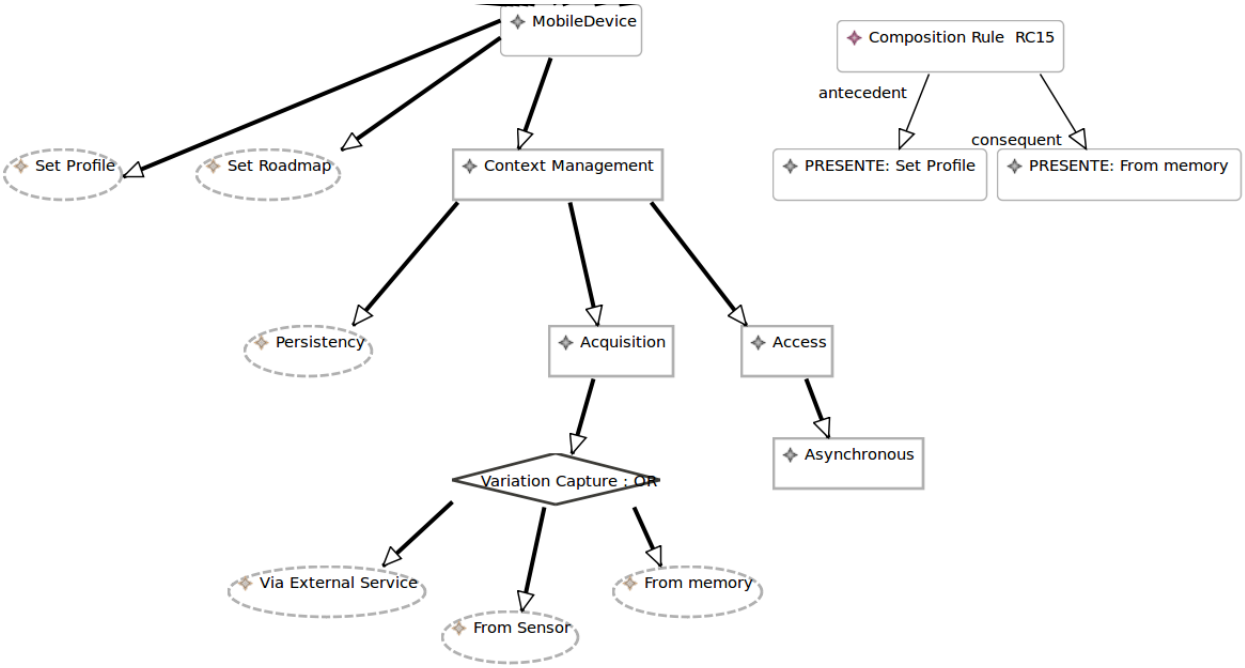


Figura 4.6: Exemplo de modelo de sistema

instâncias estão listados abaixo:

- *Context Management*.CaracteristicaPai : *Mobile Device*
- *Set Roadmap*.CaracteristicaPai : *Mobile Device*
- *Set profile*.CaracteristicaPai : *Mobile Device*
- *Persistency*.CaracteristicaPai : *Context Management*
- *Acquisition*.CaracteristicaPai : *Context Management*
- *Access*.CaracteristicaPai : *Context Management*
- *Asynchronous*.CaracteristicaPai : *Access*
- *Via External Service*.CaracteristicaPai : *Capture*
- *From Sensor*.CaracteristicaPai : *Capture*
- *From memory*.CaracteristicaPai : *Capture*

Os mesmos relacionamentos podem ser vistos como:

- *Mobile Device*.CaracteristicaFilha : *Set profile, Set Roadmap, Context Management*
- *Context Management*.CaracteristicaFilha : *Persistency, Acquisition, Access*
- *Acquisition*.CaracteristicaFilha : *OR:Capture*

(Set Profile) $\rightarrow$ *Presente opcional: From memory*	$\rightarrow$
(*verdadeiro*) $\rightarrow$ *falso*	falso

Tabela 4.1: Produto que desobedece à regra de composição

- *OR:Capture*.CaracteristicaFilha : *Via External Service, From Sensor, From Memory*
- *Access*.CaracteristicaFilha : *Asynchronous*

Essa dupla visualização permite a navegação entre os elementos em ambas as direções. Por sua vez, a regra de composição presente na Figura 4.6 pode ser entendida como uma instância do meta-modelo representado na Figura 4.4 (é importante salientar que essa notação gráfica como retângulos e losangos é incorporada no FixTure e é explicada na seção 5.4.

Tem-se uma instância de *"RegraDeComposicao"* (Regra de Composicao CR15) e duas de *"LiteralDeComposicao"* (Set Profile Presente e From memory Presente). Os relacionamentos entre si estão listados a seguir:

- *Regra de Composicao CR15*.antecedente: *Set Profile Presente*
- *Regra de Composicao CR15*.consequente: *From Memory Presente*

Essas instâncias e seus relacionamentos indicam a seguinte regra de composição: *"(PRESENTE opcional: Set Profile) implica (Presente opcional: From memory)"* que quer dizer que quando *"Set Profile"* está presente *"From memory"* também deve estar presente. Quando *"Set profile"* está presente e *"From memory"* não está, a regra de composição foi violada conforme pode ser visto na Figura 4.8.

É interessante observar que as regras de composição atuam sobre configurações de produtos. Uma configuração de produto consiste da seleção de características opcionais e da atribuição de valores para os atributos (caso as características que as possuam estejam na configuração do produto).

É possível ver um produto da LPS apresentado na Figura 4.6. Corroborando a afirmação feita no Capítulo 2, o diagrama de características contém todos os possíveis produtos da LPS. Na Figura 4.7, é possível ver que essa afirmação é válida, pois os nós destacados especificam exatamente o produto configurado.

Dado que é conhecido que o diagrama de características é uma instância do meta-modelo apresentado na Figura 4.2 e que um produto é uma instância do diagrama de características de uma LPS, pode-se então afirmar que um produto configurado é uma instância de uma instância do meta-modelo do diagrama de característica.

Neste trabalho, portanto, é dito que um produto está inconsistente quando desobedece alguma regra de composição.

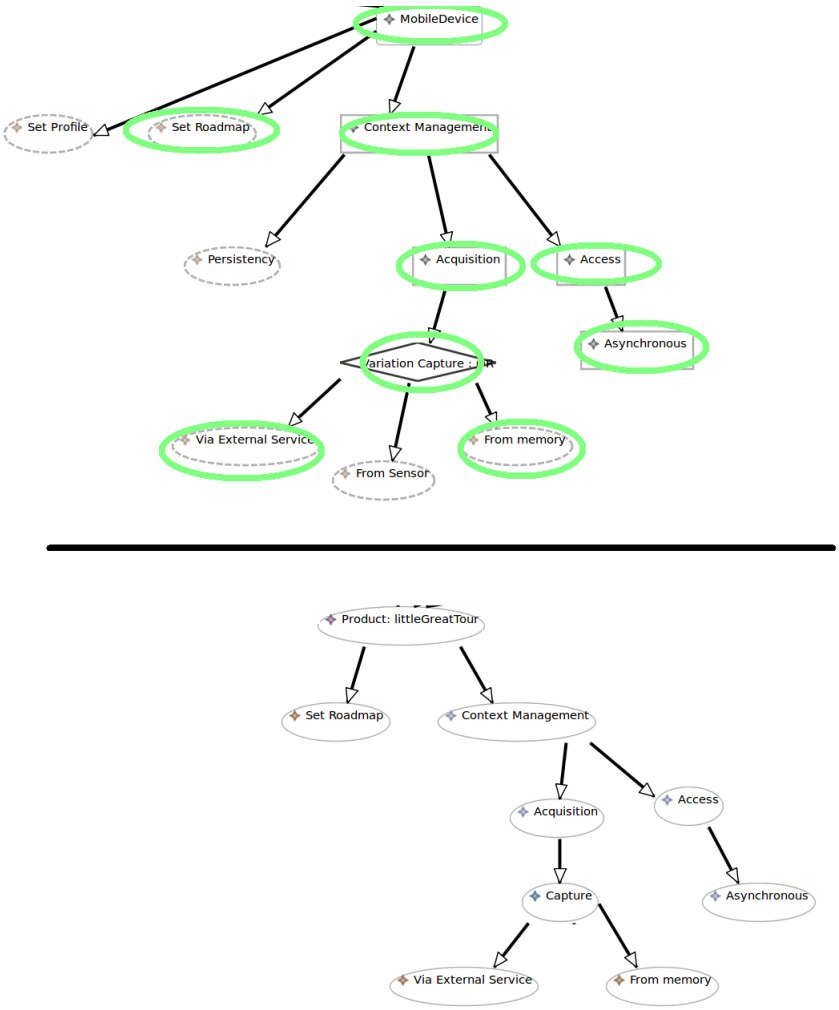


Figura 4.7: Um produto está meta descrito no diagrama de características

4.2.3 Diagrama de Contexto

Em um ambiente de uma aplicação sensível ao contexto, é necessário monitorar os valores que as entidades de contexto podem assumir. Neste caso, o meta-modelo criado para definir o diagrama de contexto apresenta uma estrutura fixa e mais simplificada quando comparada ao diagrama de características dos elementos de sistema, conforme pode ser visto na Figura 4.9.

O meta-modelo representado na Figura 4.9 é constituído das meta-classes *"Raizde-Contexto"*, *"EntidadedeContexto"* e *"InformacaodeContexto"*. O mesmo meta-modelo gráfico da Figura 4.9 pode ser visto de forma textual na Figura 4.10. Em ambas as representações, é especificado que cada *"InformacaoDeContexto"* pode estar associada com exatamente uma instância de *"EntidadeDeContexto"*, que por sua vez deve estar associada com uma instância de *"RaizDeContexto"*. O meta-modelo que representa o diagrama de contexto tem estrutura com exatamente três níveis (diferentemente do meta-modelo que representa o diagrama de características cuja profundidade é variável).

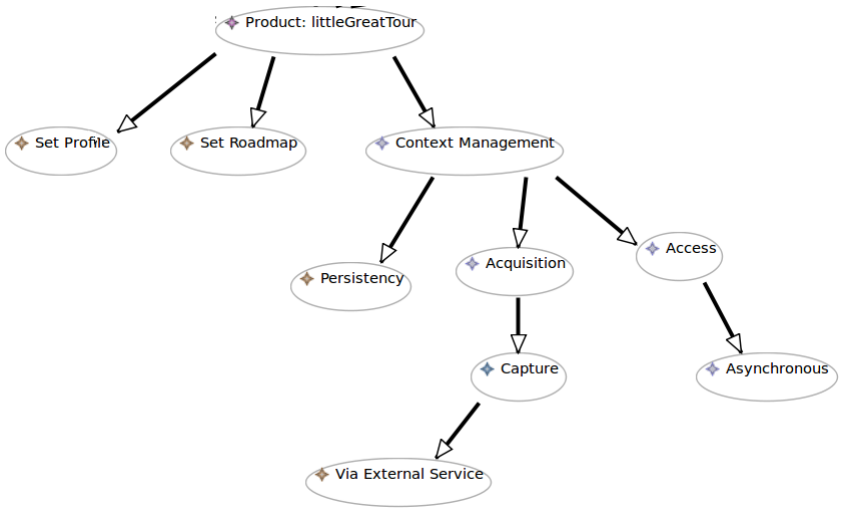


Figura 4.8: Um produto que está desobedecendo a regra de composição da Figura 4.6

Um exemplo de instância desse meta-modelo pode ser visto na Figura 4.11. Temos uma instância de "RaizDeContexto" (*Ambiente de Contexto*), três de "EntidadedeContexto" (*Bateria*, *Luminosidade* e *Localização*) e quatro de "InformaçãodeContexto" (*Carga*, *Intensidade*, *Latitude* e *Longitude*).

Daqui em diante, esse valores de contexto formam um conjunto que é chamado de *snapshot*. O snapshot que pode ser visto na Figura 4.11 é constituído do seguinte conjunto de valores: *Carga: 23*, *Intensidade: 56*, *Latitude: 25* e *Longitude: 366*. Em qualquer outro momento, esses valores podem mudar de maneira arbitrária conforme pode ser visto na Figura 4.12.

4.2.4 Regras de Contexto

As regras de contexto apresentam uma peculiaridade. Enquanto as regras de composição são formadas por uma implicação lógica (*"se ... então"*), onde tanto o antecedente como o consequente têm a mesma estrutura (seguem a mesma BNF), as regras de adaptação apresentam antecedentes e consequentes com estrutura diferenciada. A diferença nas estruturas do antecedente e do consequente visa representar o conceito das regras *"ECA"* (**event-condition-action** ou *"evento- condição-ação"*). O antecedente da regra de adaptação é formado exclusivamente de condições sobre o valor de determinada instância da classe *"InformacaodeContexto"* usando operadores relacionais ($>, <, \geq, \leq, =, \neq$). Por sua vez, o consequente é formado exclusivamente por fórmulas atômicas que tratam a remoção ou adição de características opcionais ou de alteração de valores de atributo. É conveniente chamar o antecedente de eventos e o consequente de ações.

Em outras palavras, quando uma certa combinação de valores de objetos da classe *"InformacaodeContexto"* é alcançada, certas ações são realizadas. Essas ações alteram a estrutura atual da aplicação e consistem de adições e remoções de características bem como atribuição de valores para os atributos das características.

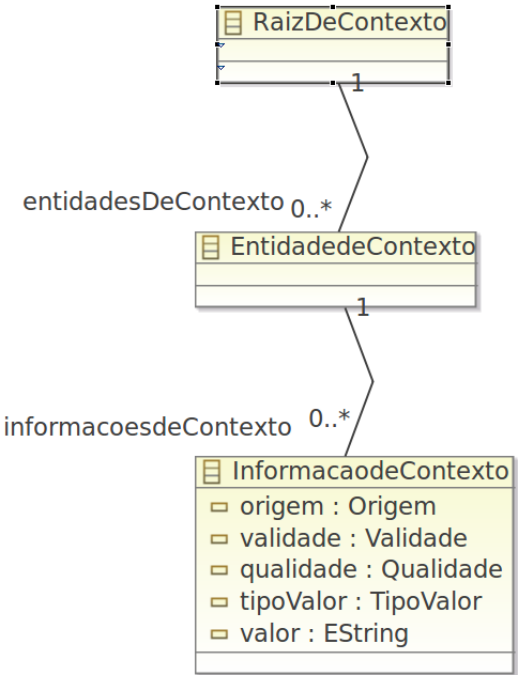


Figura 4.9: Meta-modelo do diagrama de contexto: representação gráfica

As regras de contexto são definidas usando a notação BNF da seguinte forma:

- <regradeContexto> → <evento> *implica* <acao>
- <evento> → <eventoLogico> | <eventoRelacional>
- <eventoLogico> → <evento><operadorLogico><evento>
- <eventoRelacional> → *"variaveldeContexto"*<operadorRelacional>*valor*
- <operadorLogico> → *"e"* | *"ou"*
- <operadorRelacional> → $>$ | $<$ | $\leq$ | $\geq$ | $=$ | $\neq$
- <acao> → <acaoLogico> | <literalAcao>
- <acaoLogico> → <acao><operadorAcaoLogico><acao>
- <operadorAcaoLogico> → *"e"*
- <literalAcao> → <presenca><elemento>
- <presenca> → *"presente"* | *"ausente"*
- <elemento> → *"caracteristica opcional"*
- <elemento> → *"atributo""←""valor"*

```
@gmf.node(label="nome")
class RaizDeContexto
{
    @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid",width="3")
    ref EntidadedeContexto[*]#raiz entidadesDeContexto;
}

@gmf.node(label="nome", label.pattern="Context Entity {0}")
class EntidadedeContexto
{
   ref RaizDeContexto#entidadesDeContexto raiz;
   @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid",width="3")
   ref InformacaodeContexto[*]#elementoPai informacoesdeContexto;
}


@gmf.node(label="nome")
class InformacaodeContexto
{
   attr Origem origem;
   attr Validade validade;
   attr Qualidade qualidade;
   attr TipoValor tipoValor;
   attr String valor;
   // TODO Colocar uma verificacao em EVL
   ref EntidadedeContexto[1]#informacoesdeContexto elementoPai;

}
```

Figura 4.10: Meta-modelo do diagrama de contexto: representação textual

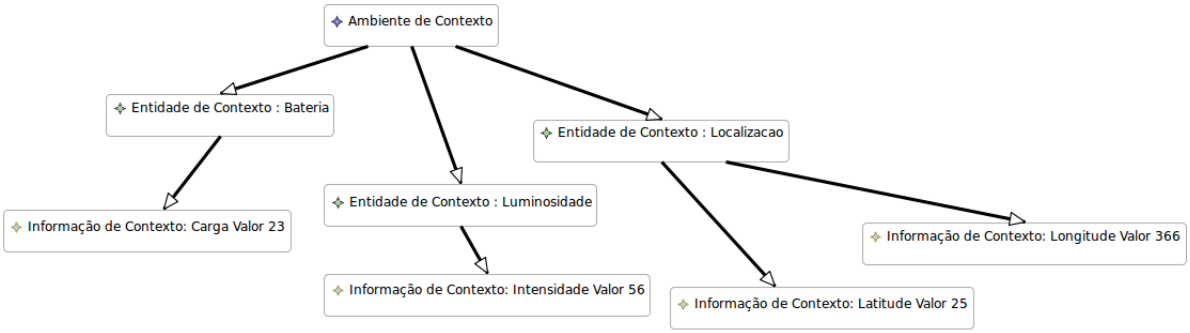


Figura 4.11: Instância do meta-modelo do diagrama de contexto

A representação das regras de contexto pode ser vista nas Figuras 4.13, 4.14 (forma textual) e 4.15 (forma gráfica).

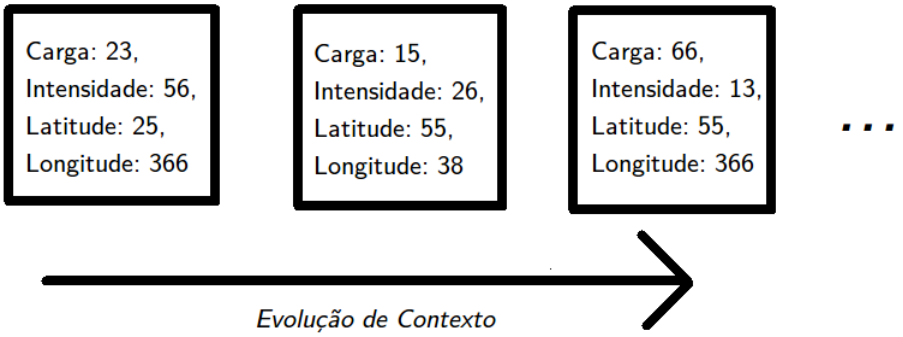


Figura 4.12: Evolução do *snapshot*

```
1  class Regra
2  {
3      attr String nome;
4      attr String conteudo;
5  }
6
7  @gmf.node(label="nome", label.pattern="Regra de Contexto {0}")
8  class RegraDeContexto extends Regra
9  {
10     @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid", label="eve
11     ref Evento evento;
12
13     @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid", label="açã
14     ref Acao acao;
15 }
16
17 @gmf.node(label="operadorLogico")
18 class EventoLogico extends Evento
19 {
20     @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid",width="3")
21     ref Evento ladoDireitoEvento;
22     attr OperadorLogico[1] operadorLogico;
23     @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid",width="3")
24     ref Evento ladoEsquerdoEvento;
25 }
26
27 @gmf.node(label="nome")
28 class EventoRelacional extends Evento
29 {
30     //@gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid")
31     ref InformacaodeContexto variaveldeContexto;
32
33     attr OperadorRelacional operadorRelacional;
34
35     attr String valor;
36 }
38 enum OperadorRelacional
39 {
40     MAIOR=0;
41     MENOR=1;
42     IGUAL=2;
43     MAIORIGUAL=3;
44     MENORIGUAL=4;
45     DIFERENTE=5;
46 }
47
```

Figura 4.13: Representação textual das regras de contexto: parte 1

Qualquer que seja a representação (textual nas Figuras 4.13 e 4.14 ou gráfica na Figura 4.15), uma instância de *"RegraDeContexto"* tem um relacionamento chamado *"evento"* com a classe *"Evento"* e um relacionamento chamado *"acao"* com a classe *"Acao"*. A classe *"Evento"* pode ser especializada por duas classes: *"EventoLogico"* e *"EventoRelacional"*. A

```
48 class Acao
49 {
50 }
51
52 @gmf.node(label="operadorAcaoLogico")
53 class AcaoLogico extends Acao
54 {
55     @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid",width="3")
56     ref Acao[1] ladoEsquerdoAcao;
57     attr OperadorAcaoLogico operadorAcaoLogico;
58     @gmf.link(target.decoration="closedarrow", color="0,0,0", style="solid",width="3")
59     ref Acao[1] ladoDireitoAcao;
60 }
61
62
63
64 @gmf.node(label="nome")
65 class LiteralAcao extends Acao
66 {
67     attr Presenca presenca;
68     ref ElementoCaracteristico elemento;
69 }
70
71 @gmf.node(label="nome")
72 class Designar  extends Acao
73 {
74     attr String valor;
75     ref Atributo atributo;
76     attr TipoValor tipoValor;
77 }
78
79 enum OperadorAcaoLogico
80 {
81    AND=0;
82 }
```

Figura 4.14: Representação textual das regras de contexto: parte 2

classe *"EventoRelacional"* define um relacionamento matemático ($>$, $<$, $\geq$, $\leq$, $=$ e $\neq$) entre uma instância de *"InformacaoDeContexto"* e um valor específico. A classe *"EventoLogico"* define dois relacionamentos com a classe *"Evento"* unidos por um operador lógico (*"E"* e *"OU"*). Dado que a própria classe *"EventoLogico"* é uma especialização da classe *"Evento"*, é possível construir aninhamento de profundidade variável. A classe *"Acao"* é especializada por três classes: *"AcaoLogico"*, *"LiteralAcao"* e *"Designar"*. A classe *"AcaoLogico"* estabelece uma relação lógica (*"E"*) entre duas instâncias da classe *"Acao"* (semelhantemente a classe *"EventoLogico"*, esta classe permite a construção de aninhamento de operadores lógicos de profunidade variável). A classe *"LiteralAcao"* estipula sobre a adição ou remoção de uma característica não-mandatória em um produto. A classe *"Designar"* atribui valores específicos aos atributos das características de um produto. As especializações *"LiteralAcao"* e *"Designar"* da classe *"Acao"* especificam as alterações que são realizadas sobre um produto.

Na Figura 4.16 é possível ver uma instância do meta-modelo apresentado nas Figuras 4.9 e 4.10. Essa regra tem o seguinte significado: *"((Carga $>$ 50) OR (Intensidade $\leq$ 40)) implica ((Potencia $\leftarrow$ 45) AND (Remover Automático) AND (Inserir Trailer))"*. Se a carga da bateria for maior que 50 ou se a intensidade luminosa for menor ou igual que 40, então, remova o câmbio automático, insira o trailer e atribua 45 para a potência. Nota-se que o evento trata somente de valores de contexto ao passo que a ação trata somente dos elementos do diagrama de características, conforme fora dito anteriormente.

É importante observar que certos *snapshots* podem ativar ações quando as condições de evento são satisfeitas. Por exemplo, o seguinte *snapshot* [*Carga: 23*, *Intensidade: 20*, *Latitude: 25* e *Longitude: 366*] dispara a regra de adaptação *"((Carga $>$ 50) OR (Intensidade $\leq$ 40)) implica ((Potencia $\leftarrow$ 45) AND (Remover Automático) AND (Inserir Trailer))"*. Logo, no momento que um *snapshot* satisfaz as condições de evento de uma regra de adaptação, as ações da referida regra de adaptação são executadas. Consequentemente, com a mudança do *snapshot*, possivelmente outras regras de contexto são disparadas.

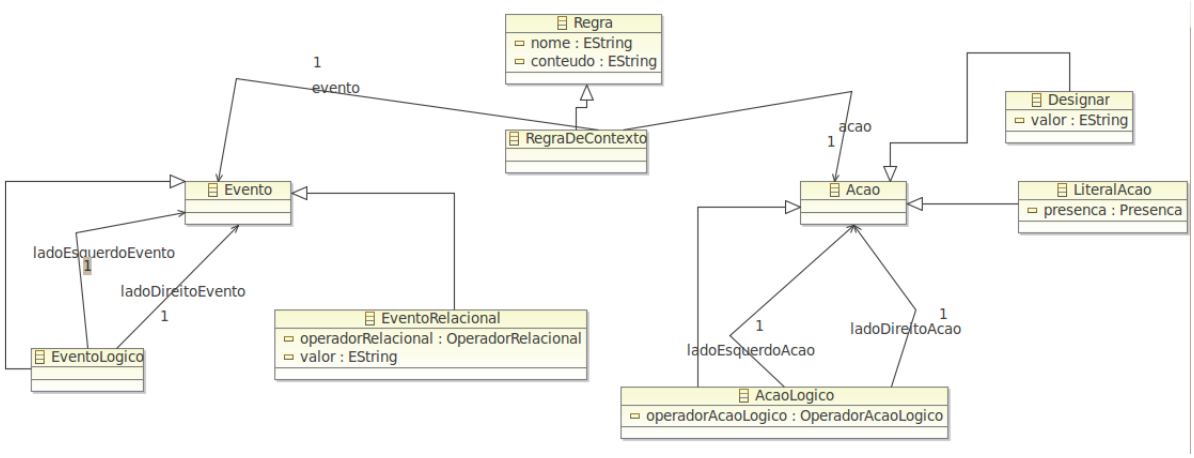


Figura 4.15: Representação gráfica das regras de contexto

Em um ambiente sensível ao contexto, que está sujeito a grandes variações de valores de contexto (como um ambiente móvel por conta da mobilidade que o usuário possui), o produto corrente sofre várias alterações ao longo do tempo.

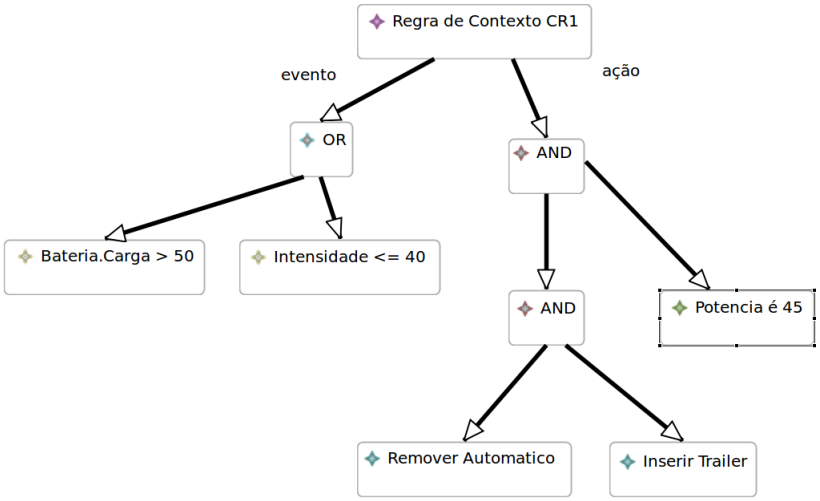


Figura 4.16: Instância do meta-modelo das regras de contexto

4.2.5 Regras de boa formação

Conforme dito no início da seção 4.1, os quatro meta-modelos (diagrama de características, regras de composição, diagrama de contexto e regras de contexto) possuem regras de boa formação que definem diretrizes para a construção de elementos corretos e consistentes. Pode-se citar um exemplo simples a seguir. Em um diagrama de características, uma característica mandatória não deve ser modelada como filha de uma característica opcional. Como a característica opcional pode não estar presente em alguns produtos, isso impossibilitaria a presença da característica mandatória que é a sua filha. Esta é uma regra de boa formação e ela precisa ser especificada de alguma forma.

Parte das regras de boa formação que este presente trabalho utiliza foram listadas

na tese de doutorado de (MARINHO, 2012). Nessa tese de doutorado, as regras de boa formação foram formalizadas usando OCL. Por possuir uma sintaxe parecida com OCL e por estar disponível para modelos *Ecore* (BUDINSKY, 2004), a linguagem Epsilon Validation Language (EVL)(BUDINSKY, 2004) foi escolhida para especificar as regras de boa formação. EVL oferece uma linguagem de restrições similar a OCL (EVL, 2012). Contudo, há algumas adições como suporte às dependências entre restrições e às especificações de correções em caso de desobediência às restrições.

A estrutura de uma restrição em EVL segue o formato apresentado na Listagem 4.2:

```
1  context Caracteristica
2  {
3    constraint NomesDeAtributosUnicos
4    {
5      guard: self.atributo
6                  ->forAll(x | x.nome<>'' and x.nome.trim()<>'')
7      check: self.atributo
8                 ->forAll(x | self.atributo
9                   ->forAll(y | x<>y implies x.nome<>y.nome))
10   }
11 }
```

(4.2)

A estrutura das restrições consiste de *"context"* (que define à qual meta-classe do meta-modelo essa restrição se aplica), *"guard"* (uma pré-condição que se for avaliada como falsa impede a checagem do elemento *"check"*) e *"check"* (que corresponde à checagem em si). No exemplo, é verificado explicitamente se uma característica não possui dois atributos com o mesmo nome. Essa verificação é realizada para todas as instâncias da meta-classe *"Característica"* do meta-modelo. A condição *"guard"* verifica para todos os atributos da instância de *"Característica"* se os nomes dos atributos não são vazios. A condição *"check"* se não existem dois atributos com mesmo nome.

Nesta direção, neste trabalho é realizada a tradução das regras de boa formação apresentadas e reunidas em (MARINHO; ANDRADE; WERNER, 2011). A listagem completa de todas as restrições especificadas em EVL pode ser integralmente consultada no site do MobiLine (MOBILINE, 2012)

Diante disso, é dito que um modelo é bem formado se ele obedece às regras de boa formação.

4.3 Conclusões

Neste capítulo, foram mostrados os meta-modelos referentes a uma LPSSC. Os meta-modelos apresentados aqui contém melhorias em relação aos meta-modelos apresentados em (MARINHO, 2012) nos quais este trabalho se baseou. Entre as melhorias pode-se citar tanto o aprofundamento e melhor definição dos meta-modelos como a representação das instâncias dos produtos da LPS.

Com essa modelagem, os engenheiros de software podem representar os diagramas de características usando a classificação de características mandatórias, opcionais e pontos de variação. Também podem representar as regras de composição que podem ser claramente compreendidas como restrições sobre o diagrama de características modelado. Pelo lado contextual da LPSSC, o meta-modelo diagrama de contexto representa as informações contextuais relevantes. Por fim, a adaptabilidade é especificada através de regras de contexto. Dado esses quatro meta-modelos, o engenheiro de software poderá instanciar o diagrama de características na forma de um produto.

Uma vez que os meta-modelos foram definidos no presente capítulo, o passo seguinte é definir operações em cima desses meta-modelos a fim de oferecer a verificação automática de tais meta-modelos.

O próximo capítulo trabalha com a definição dessas operações. Como exemplo dessas operações, pode-se citar a verificação se uma configuração do produto respeita o diagrama de características de que se originou bem como se respeita as regras de composição estabelecidas. Essas operações também são consideradas quando um produto já configurado sofre uma adaptação.

5 MECANISMO DE VERIFICAÇÃO AUTOMÁTICA

Neste capítulo são expostas a ferramenta FixTure e a automatização do mecanismo de verificação PRECISE (MARINHO, 2012). Parte das operações executadas nesse mecanismo é executado em cima dos meta-modelos definidos no Capítulo 4. Porém, outra parte das operações são mais complexas. Para a execução destas operações, é exigido que os meta-modelos sejam transformados. No presente trabalho, os meta-modelos são transformadas em fórmulas proposicionais. Uma vez consideradas essas questões, é exibida a forma como cada uma das fases do PRECISE foi implementada.

A Seção 5.1 expõe um exemplo recorrente ao longo deste capítulo facilitando assim a leitura deste. A Seção 5.2 justifica a necessidade da transformação mencionada e mostra detalhadamente como realizar esta transformação. A Seção 5.3 mostra a implementação do PRECISE com destaque para a Sub-Seção 5.3.5 onde são discutidas duas abordagens para a simulação. Em seguida, a Seção 5.4 apresenta a ferramenta FixTure e suas funcionalidades bem como a geração do grafo de simulação. Por fim, a Seção 5.5 apresenta as conclusões sobre o que é apresentado no presente capítulo.

5.1 Exemplo

Durante o restante do capítulo para exemplificar a automatização do processo, será usado os exemplos das Figuras 5.1 e 5.2.

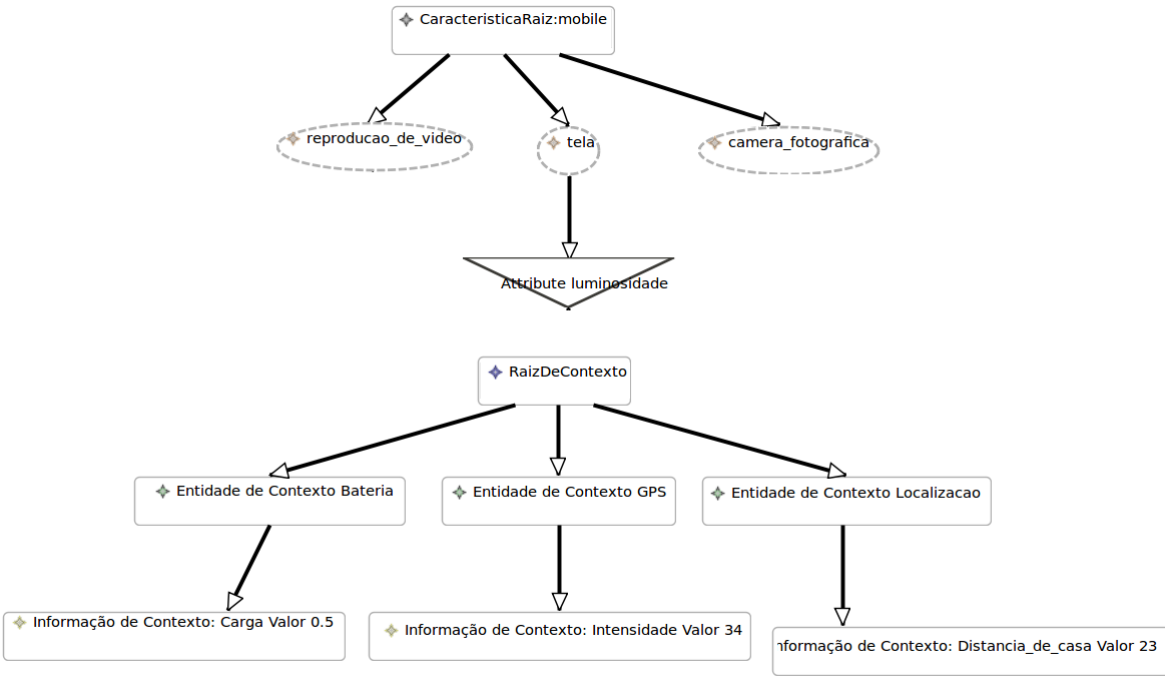


Figura 5.1: Diagramas de características e de contexto do exemplo

Na Figura 5.1, o diagrama de características é formado pelas seguintes instâncias

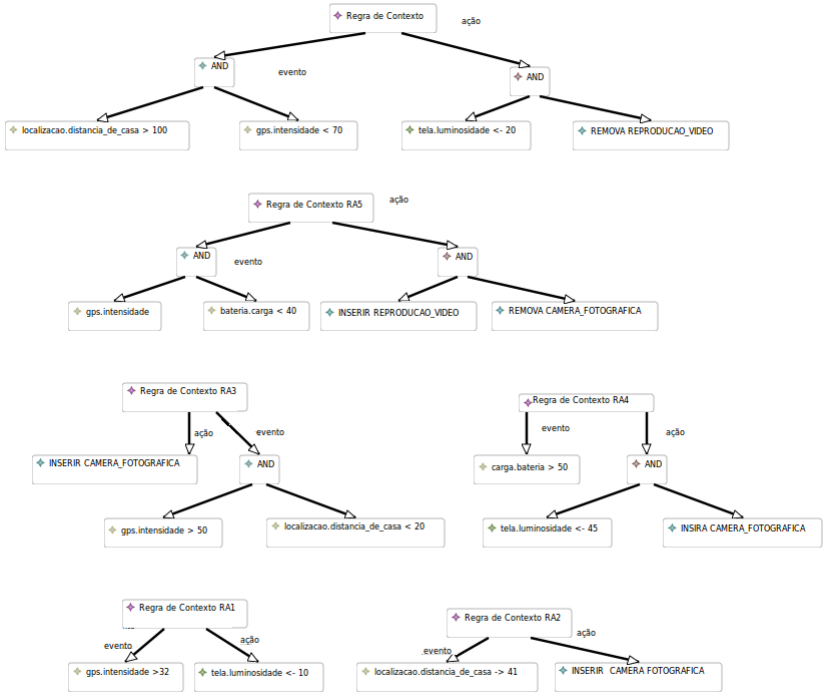


Figura 5.2: Regras de contexto do exemplo

de *"CaracteristicaOpcional"*: *camera_fotografica*, *reproducao_de_video* e *tela*. Associado a *tela*, há uma instância de *"Atributo"* chamada *luminosidade*.

Na Figura 5.1, o contexto relevante é formado pelas seguintes instâncias de *"EntidadeDeContexto"*: *bateria*, *gps* e *localizacao*. Para cada uma destas instâncias, há uma instância de *InformacaoDeContexto*: *carga*, *intensidade* e *distancia_de_casa*, respectivamente.

As regras de contexto detalhadas na Figura 5.2 são:

- $RA1$ gps_sinal $>$ 32 $\rightarrow$ (luminosidade $\leftarrow$ 10)
- $RA2$ distancia$_de_$casa $>$ 41 $\rightarrow$ INSIRA $CAMERA\_FOTOGRAFICA$
- $RA3$ gps_sinal $>$ 50 AND distancia$_de_$casa $<$ 20 $\rightarrow$ REMOVA $REPRODUCAO\_VIDEO$
- $RA4$: $carga\_bateria > 50 \rightarrow$ (luminosidade $\leftarrow$ 45) AND (INSIRA $CAMERA\_FOTOGRAFICA$)
- $RA5$: $sinal\_gps > 25$ AND $carga\_bateria < 40 \rightarrow$ (INSIRA $REPRODUCAO\_VIDEO$) AND (REMOVA $CAMERA\_FOTOGRAFICA$)
- $RA6$: $distancia\_de\_casa > 100$ AND $sinal\_gps < 70 \rightarrow$ (luminosidade $\leftarrow$ 20) AND (REMOVA $REPRODUCAO\_VIDEO$)

5.2 Representação de uma LPSSC como fórmulas proposicionais

Antes de mostrar a implementação do PRECISE proposta em (MARINHO, 2012), para o qual o presente trabalho dá suporte, é necessário mostrar as transformações que são

executadas.

Embora o meta-modelo definido na seção 4.1 permita realizar operações sobre o mesmo (*reasoning*), neste trabalho tanto o modelo do sistema como as instâncias deste são transformados em fórmulas booleanas a fim de submetê-los a um resolvedor de fórmulas booleanas do tipo Diagrama de decisão binária (DDB). A transformação em fórmulas proposicionais consiste em declarar todos os elementos (diagramas e regras) bem como os relacionamentos entre os mesmos usando lógica proposicional. Este processo de transformação é realizado em cima do diagrama de característica e do de contexto e das regras de composição e das de contexto.

Nessa transformação, necessária para a execução das fases do PRECISE, as características (tanto opcionais como mandatórias bem como os pontos de variação) são representadas como variáveis "*booleanas*" (possuem valor 0 ou 1). Elas só recebem algum valor (verdadeiro ou falso) quando analisadas em uma configuração de produto, pois estas variáveis recebem valor verdade "verdadeiro" quando estão presentes na configuração do produto e "falso" quando não estão.

Pela própria definição de característica mandatória, as variavéis que representam as características mandatórias sempre têm valor 1. No entanto, para uma característica opcional, por exemplo, *texto*, é necessário criar a variável booleana V_{texto} que pode assumir os valores 0 ou 1.

Os atributos das características têm a sua presença determinada pela presença da característica ao qual se ligam: se a característica está presente, obrigatoriamente o atributo também está. De forma semelhante, se a característica não está presente, então o atributo também não está.

Como mostrado na Figura 4.4, as regras de composição (e de contexto também) são representadas em forma de árvore no sentido de acomodar a especificação BNF. Algumas das classes presentes nessa representação representam uma folha na árvore, isto é, não possuem mais filhos. Essas classes são *"ExpressaoRelacional"* e *"LiteralComposicao"*. Para essas classes é dado o nome de fórmulas atômicas.

Duas ou mais fórmulas atômicas podem referenciar o mesmo atributo através dos operadores matemáticos relacionais: $>$, $<$, $\geq$, $\leq$, $=$ e $\neq$. Quando isso acontece, dependendo de quais operadores matemáticos relacionais tenham sido usados, diferentes campos de variação podem ter que ser criados. Na Seção 5.3.2, é abordada essa problemática de múltiplas fórmulas atômicas referenciarem o mesmo atributo..

Por esse motivo, os atributos não são diretamente referenciados: em vez de criar uma variável booleana para cada atributo, são criadas variáveis booleanas para as fórmulas atômicas (presentes nas regras de composição) que fazem referência a algum atributo.

Uma regra de composição é uma implicação lógica (*se ... então*) contendo várias fórmulas atômicas ligadas por instâncias dos elementos *"ExpressaoLogica"*. Essa classe, como dito anteriormente, funciona como conector lógico usando os operadores de conjunção (*E*) e disjunção (*OU*). Para cada fórmula atômica que faz referência a um atributo é criado uma va-

riável booleana.

Para exemplificar este processo, será considerada a Figura 5.3. Nesta figura, há uma LPS com três nós atributos $m_1.attr_1$, $m_2.attr_1$ e $o_1.attr_1$. Ainda na mesma figura, podem-se ver as seguintes fórmulas atômicas: a_1*: m1.attr*$_1 > 30$, a_2*: not o2* e a_3*: m2.attr*$_1 < 23$. As fórmulas atômicas referenciam dois atributos *m1.attr*$_1$ e *o1.attr*$_1$. Por isso, para cada uma dessas duas fórmulas atômicas (a_1 e a_3), são criadas uma variável booleana (Va_1 e Va_3, respectivamente). Os campos de variação do valor do atributo podem surgir devido às diferentes formas de combinação dos operadores matemáticos relacionais.

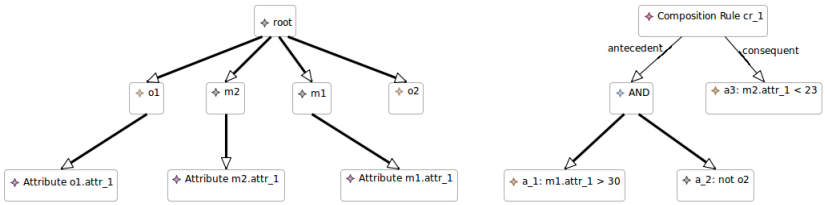


Figura 5.3: Atributos referenciados: $m_1.attr_1$ e $m_2.attr_1$

Com isso, a LPS representada na Figura 5.3 tem a seguinte representação simbólica:

$$Vroot : [1], Vm_1 = [1], Vo_1 = [0/1], Vo_2 = [0/1], Va_1 = [0/1], Va_3 = [0/1]$$

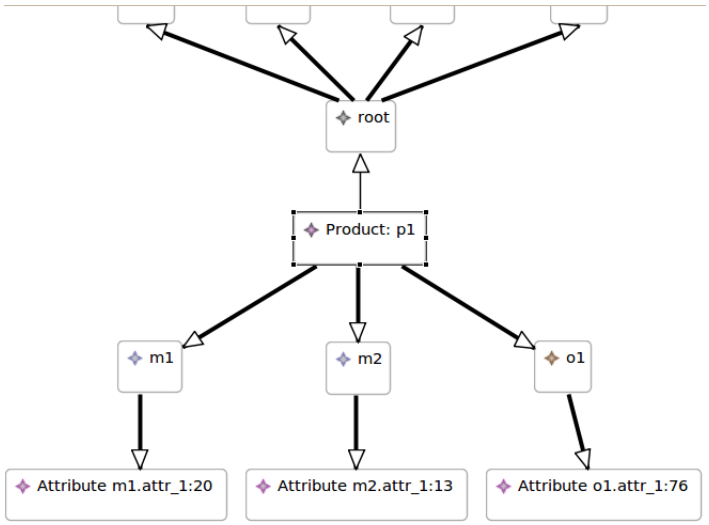


Figura 5.4: Uma possível configuração

Com essa representação, o produto dessa LPS exibido na Figura 5.4 terá a seguinte representação simbólica

$$Vroot : [1], Vm_1 = [1], Vo_1 = [1], Vo_2 = [0], Va_1 = [0], Va_3 = [1]$$

Dado que a regra de composição é uma implicação lógica (uma fórmula booleana), a transformação desta em fórmula booleana é direta.

Considere a regra de composição cr_1 exibida na Figura 5.3, logo:

$$cr_1 : ((m1.attr_1 > 30) \wedge (not\ o_2)) \rightarrow (m2.attr_1 < 23)$$

Para cada fórmula atômica que lida com atributo é criada uma variável booleana. Assim, a regra de composição CR_1 assume a seguinte configuração:

$$Vcr_1 : ((Vaf_1) \wedge (not\ Vo_2)) \rightarrow (Vaf_3)$$

Observe que as variáveis booleanas criadas para cada uma das características e pontos de variação são usadas diretamente na transformação da regra de composição.

É interessante notar que ao adotar essa representação simbólica a regra de composição é simplificada pois retiram-se da mesma os operadores relacionais ($>$, $<$, $\geq$, $\leq$, $=$ e $\neq$). Além disso, a adoção dessa representação permite que seja obtida uma vantagem durante a simulação, pois em vez de se vasculhar o espectro de valores dos tipos do atributo (por exemplo, para um atributo do tipo *"Inteiro"* o espectro de valores consiste de todos os números inteiros suportados pela arquitetura do processador), serão usados apenas os valores booleanos *"Verdade"* ou *"Falso"*.

Um ponto interessante dessa representação simbólica surge quando há duas ou mais fórmulas atômicas referenciando o mesmo atributo conforme pode ser visto na Figura 5.5.

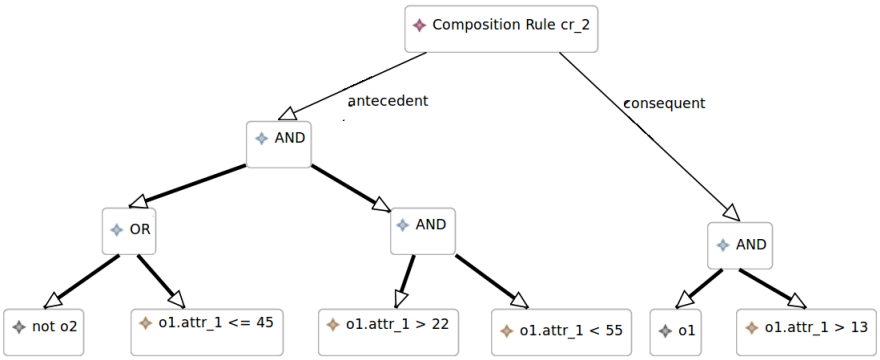


Figura 5.5: Fórmulas referenciando o mesmo atributo

Neste caso, as seguintes variáveis booleanas são consideradas: $af_1 : o1.attr_1 \leq 45$, $af_2 : o1.attr_1 > 22$, $af_3 : o1.attr_1 > 13$ e $af_4 : o1.attr_1 > 5$. Portanto, existem restrições para os valores que af_1, af_2 e af_3 podem assumir simultaneamente. Por exemplo, é impossível af_1 e af_2 terem valores igual a falso simultaneamente, dado que qualquer inteiro ou é menor ou igual a 45 ou é maior que 22. Outro exemplo consiste no par af_2 e af_3, pois é impossível af_2 ter valor igual a verdadeiro e, ao mesmo tempo, af_3 ter valor igual a falso já que quando um valor inteiro é maior que 22 (af_2), consequentemente, esse valor é maior que 13 também (af_3).

Essas restrições devem ser consideradas durante a verificação da satisfazibilidade da conjunção total das regras de composição.

Se por um lado é fácil identificar as restrições quando apenas duas fórmulas atô-

Variável booleana criada	Fórmula representada pela variável
af_1	$o1.attr_1 \leq 45$
af_2	$o1.attr_1 > 22$
af_3	$o1.attr_1 > 13$
af_4	$o1.attr_1 < 55$

Tabela 5.1: Fórmulas atômicas que referenciam o mesmo atributo $o_1.attr_1$

micas são consideradas, o mesmo não pode ser dito quando várias fórmulas atômicas referenciando o mesmo atributo são levadas em consideração. Assim, algum procedimento deve ser criado para identificar essas restrições entre as fórmulas atômicas que referenciam atributos.

Tal procedimento é inicialmente ilustrado nas tabelas 5.1 e 5.2 , e depois formalizado no algoritmo 1.

Na Tabela 5.1, são listadas as fórmulas atômicas que referenciam o mesmo atributo. Em tal tabela, é possível ver que há operadores relacionais matemáticos diferentes ($>$, $<$ e $\leq$) com valores diferentes.

Em seguida, é preciso identificar quais as possíveis combinações entre as variáveis booleanas af_1, af_2, af_3 e af_4. Uma abordagem inicial seria testar todos os valores possíveis que $o1.attr_1$ pode assumir e então ver quais são as combinações de valores para as variáveis supracitadas af_1, af_2, af_3 e af_4. Essa abordagem não é eficiente, pois a quantidade de valores que a variável $o_1.attr_1$ pode assumir é alta (no caso de uma variável do tipo *"Inteiro"* ou *"Real"*, seria exatamente a quantidade de valores inteiros ou reais que o processador consegue representar). Esse problema foi originalmente identificado em (SAMA et al., 2010). A tabela 5.2 mostra como é feita a identificação das restrições no presente trabalho. A ideia é estimular as variáveis *"booleanas"* af_1, af_2, af_3 e af_4 de forma a identificar todas as combinações. *A priori*, como há quatro variáveis booleanas, há, então, 2^4 combinações possíveis (contudo, algumas dessas 16 combinações não são possíveis conforme explicado anteriormente). O intuito, então, é descobrir quais combinações são, de fato, válidas. A ideia do algoritmo descrito no presente trabalho consiste em escolher o menor conjunto de valores que vão estimular as variáveis booleanas criando apenas as combinações válidas. Na Tabela 5.2, são identificados nove valores para o atributo $o_1.attr_1$.

A criação consiste em atribuir valores inteiros específicos para a variável $attr_1$ para descobrir todas as possíveis combinações entre as fórmulas atômicas. O algoritmo 1 mostra como criar esses valores específicos. O algoritmo 1 inicialmente ordena todos os valores que referenciam o mesmo atributo em questão. No caso da Tabela 5.1, os valores ordenados produzem o seguinte conjunto: 13, 22, 45 e 55. É possível observar que, considerando estes 4 valores, há 5 intervalos: menor que 13, entre 13 e 22, entre 22 e 45, entre 45 e 55 e maior que 55. Esses valores podem ser criados tomando a média entre dois valores consecutivos (por exemplo, entre 13 e 22, a média seria 17,5) e a metade do primeiro e o dobro do último valor conforme se observa nas linhas 4, 5, 6 e 7 do algoritmo 1.

Observando a Figura 5.6, é mais fácil entender o porquê da escolha desses valores.

Valor do atributo $o1.attr_1$	valor de af_1	valor de af_2	valor de af_3	valor de af_4
2	verdadeiro	falso	falso	verdadeiro
5	verdadeiro	falso	falso	falso
9	*verdadeiro*	*falso*	*falso*	*falso*
13	*verdadeiro*	*falso*	*falso*	*falso*
18	verdadeiro	falso	verdadeiro	falso
22	*verdadeiro*	*falso*	*verdadeiro*	*falso*
33	verdadeiro	verdadeiro	verdadeiro	falso
45	*verdadeiro*	*verdadeiro*	*verdadeiro*	*falso*
90	falso	verdadeiro	verdadeiro	falso

Tabela 5.2: Possíveis combinações das fórmulas atômicas que referenciam o mesmo atributo $attr_1$

Em vez de escolher todos os possíveis valores que uma variável pode assumir, são escolhidos apenas três valores para variável booleana. A justificativa pode ser encontrada observando a seguinte expressão relacional: $x < 30$. Sabe-se que para qualquer valor menor que 30, a expressão relacional $x < 30$ é verdadeira. Para um valor igual a 30, a $x < 30$ continua falsa. Um valor maior que 30 faz a expressão $x < 30$ ser falsa. Considerando apenas os três valores (29, 30 e 31), a expressão relacional $x < 30$ produz todos os possíveis valores. A ideia no algoritmo 1 utiliza esse princípio e ainda considera que variáveis *"booleanas"* referenciando o mesmo atributo. Por exemplo, ao considerar agora duas expressões relacionais ($x < 30$ e $x = 10$), é necessário estimular a variável x de mais formas. Como cada expressão relacional precisa ser estimulada por três valores, *a priori*, seriam necessários seis valores para a variável x, porém uma pequena otimização pode ser feita se consideramos esses seis valores estimulam ambas as expressões relacionais. Os valores para a variável x quando é considerada a expressão relacional ($x < 30$) seriam 29, 30 e 31 e, quando considerando a expressão relacional $x = 10$, seriam 9, 10 e 11. Em vez de considerar os valores para x tomando as expressões relacionais separadamente, o algoritmo 1 aborda os intervalos entre esses valores (10 e 30) de forma a verificar todas as combinações possíveis.

Algoritmo 1 Cria os novos valores a serem inseridos no intervalo de teste

```
1: Entrada: valores ordenados V
2: Saída: novos Valores nV
3: nV.add(V[0]/2)
4: for all valor_i f in V do
5:     nV.add(valor_i)
6:     nV.add((valor_i + valor_{i+1}/2))
7: end for
8: nV ← V_ultimo × 2
9: return nV
```

No Algoritmo 1, são coletadas todas as fórmulas atômicas relacionais que referenciam o mesmo atributo (se mais de um atributo é referenciado mais de uma vez, o algoritmo é executado para cada atributo referenciado mais de uma vez nas regras de composição).

valor do atributo $o1.attr_1$	Valores das variáveis booleanas para este valor de atributo
2	$af_1 \wedge \neg af_2 \wedge \neg af_3 \wedge af_4$
5	$af_1 \wedge \neg af_2 \wedge \neg af_3 \wedge \neg af_4$
9	$af_1 \wedge \neg af_2 \wedge \neg af_3 \wedge af_4$
13	$af_1 \wedge \neg af_2 \wedge \neg af_3 \wedge af_4$
18	$af_1 \wedge \neg af_2 \wedge af_3 \wedge \neg af_4$
22	$af_1 \wedge \neg af_2 \wedge af_3 \wedge \neg af_4$
33	$af_1 \wedge af_2 \wedge af_3 \wedge \neg af_4$
45	$af_1 \wedge af_2 \wedge af_3 \wedge \neg af_4$
90	$\neg af_1 \wedge af_2 \wedge af_3 \wedge \neg af_4$

Tabela 5.3: Possíveis combinações das fórmulas atômicas que referenciam o mesmo atributo $attr_1$

A ideia deste algoritmo é verificar os possíveis valores das variáveis booleanas (criadas para representar os atributos) usando valores intermediários. Por exemplo, para as fórmulas atômicas relacionais representadas na Tabela 5.1 temos os seguintes valores iniciais ordenados: 5, 13, 22 e 45. Os valores intermediários seriam: 2, 9, 18, 33 e 90. Em seguida para um cada desses valores, as variáveis booleanas são testadas. O resultado é apresentado na Tabela 5.2.

Por fim, cada linha da tabela 5.2 é transformada em uma fórmula proposicional. A linha que contém o valor 2 para $o_1.attr_1$ se transformaria na seguinte fórmula: $af_1 \wedge \neg af_2 \wedge \neg af_3 \wedge af_4$ o que quer dizer que as variáveis af_1 e af_4 devem ter valor booleano *"Verdade"* e af_1 e af_4, valor booleano *"Falso"*. A transformação linha a linha é mostrada na Tabela 5.3.

Por fim, faz-se necessário aplicar o operador de *"Ou Exclusivo"* no sentido de dizer ***que uma e apenas uma*** destas restrições deve ser respeitada. A disjunção exclusiva total é: $(af_1 \wedge \neg af_2 \wedge \neg af_3 \wedge af_4) \oplus (af_1 \wedge \neg af_2 \wedge \neg af_3 \wedge \neg af_4) \oplus (af_1 \wedge \neg af_2 \wedge af_3 \wedge \neg af_4) \oplus (af_1 \wedge af_2 \wedge af_3 \wedge \neg af_4) \oplus (\neg af_1 \wedge af_2 \wedge af_3 \wedge \neg af_4)$.

Pode-se ver que se uma LPSSC contém f fórmulas atômicas, o algoritmo 1 precisa de $\mathcal{O}(2 * f + 1)$ operações para criar os valores necessários para a cobertura.

É necessário que este algoritmo gere todas as combinações possíveis nas quais as regras de contexto podem ser ativadas simultaneamente. Inicialmente, para um conjunto *RA* com $|RA|$ regras de contexto, existem até $2^{|RA|} - 1$ combinações possíveis (considerando até quando a regra é ativada isoladamente). Esse é exatamente o número de subconjunto de tamanho maior que 1. Como foi visto, algumas combinações de regras de contexto podem não existir pois seus eventos não são simultaneamente ativáveis (não existe um *"snapshot"* que ative o evento das duas regras ao mesmo tempo). Portanto, o número de combinações possíveis será $\mathcal{O}(2^{|RA|} - 1)$. Para garantir a cobertura dessas $\mathcal{O}(2^{|RA|} - 1)$ combinações possíveis, o algoritmo 1 cria valores que fazem cada uma das fórmulas atômicas criadas terem o valor verdade como *"verdadeiro"* e como *"falso"*. É necessário lembrar que o algoritmo 1 trabalha somente com as fórmulas atômicas referentes a um atributo. Para cada conjunto de fórmulas atômicas criadas para cada atributo, o algoritmo é repetido.

Com isso, todas as possíveis combinações de fórmulas atômicas são geradas. Uma vez geradas, elas são submetidas às fórmulas proposicionais para então assinalar quais regras de contexto são ativadas simultaneamente.

Na Figura 5.6, esse processo fica mais visível onde é possível ver que são criados intervalos de valores para as fórmulas atômicas de forma a explorar todas as combinações possíveis.

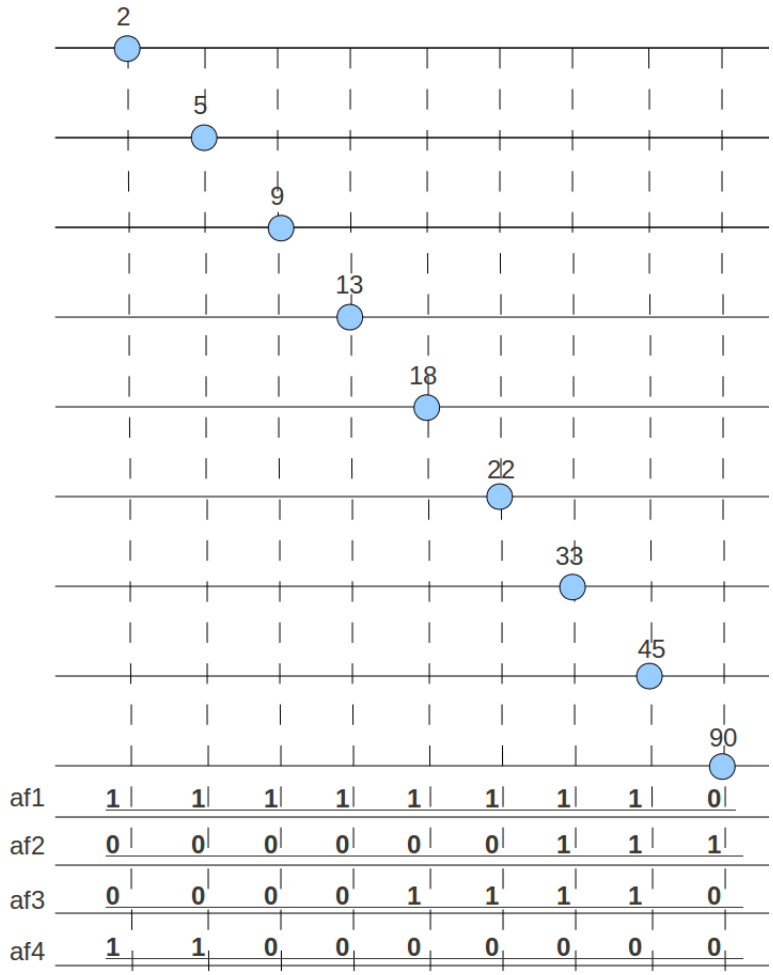


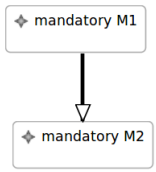
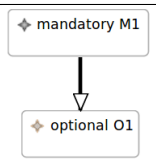
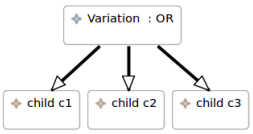
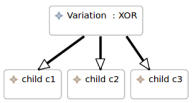
Figura 5.6: Cobertura dos valores para as fórmulas atômicas criadas para um mesmo atributo

É necessário também transformar os relacionamentos de opcionalidade, alternância (exclusiva ou não) e obrigatoriedade como fórmulas proposicionais. Esse processo é descrito em parte em (BENAVIDES; SEGURA; RUIZ-CORTÉS, 2010). É interessante observar que (BENAVIDES; SEGURA; RUIZ-CORTÉS, 2010) não considerou os atributos na sua transformação. Como o presente trabalho oferece atributos no modelos de características, a transformação destes teve de ser detalhada como, de fato, foi feita na presente seção.

A tabela 5.4 mostra as transformações que podem ser executadas.

Por fim, também é necessário transformar o diagrama de contexto e as regras de contexto. Para os fins de implementação do PRECISE, são representadas apenas as instâncias da classe *"InformacaoDeContexto"* pois são nos objetos desta classe que ficarão armazenados

Tabela 5.4: Mapeamento de modelo de sistema para lógica proposicional (adaptado de (BENAVIDES; SEGURA; RUIZ-CORTÉS, 2010))

Relacionamento		Mapeamento
mandatória		$m_2 \rightarrow m_1$
opcional		$m_2 \leftrightarrow m_1$
variação opcional		$v \rightarrow (c_1 \vee c_2 \vee c_3)$
variação exclusiva		$(c_1 \leftrightarrow (v \wedge \neg c_2 \wedge \neg c_3)) \wedge$ $(c_2 \leftrightarrow (v \wedge \neg c_1 \wedge \neg c_3)) \wedge$ $(c_3 \leftrightarrow (v \wedge \neg c_1 \wedge \neg c_2))$

os valores de contexto que realmente influenciarão a adaptação do produto. Como dito na seção 4.2.3, o conjunto de valores de contexto é chamado de *"snapshot"*. Na FixTure, o conjunto dessas instâncias formará o *"snapshot"* que é basicamente uma representação momentânea do contexto atual.

As regras de contexto, conforme pode ser visto na Figura 4.16, são modeladas em árvore. Os nós folha de uma regra de contexto são instâncias da classe *"EventoRelacional"*. Essa classe referencia uma instância da classe "InformacaoDeContexto". Semelhantemente aos atributos de características, essas instâncias não serão diretamente transformadas em variáveis booleanas. Em verdade, serão transformadas em variáveis booleanas as instâncias da classe *"EventoRelacional"* pois elas é que fazem uso, de fato, das instâncias da classe *"InformacaoDeContexto"*. Assim, para cada instância da classe *"EventoRelacional"* criaremos uma variável booleana.

5.3 Implementação do PRECISE

Em (MARINHO; ANDRADE; WERNER, 2011) e (MARINHO, 2012), é proposto o processo de verificação para modelos de características de aplicações móveis e sensíveis ao contexto chamado PRECISE. O presente trabalho oferece uma implementação desse processo.

O PRECISE é composto de cinco fases:

1. Verificar a boa formação de diagramas de características e de contexto: as regras de boa formação do diagrama de características e de contexto são avaliadas;
2. Verificar a boa formação das regras de composição e de adaptação: as regras de boa formação das regras de composição e de contexto são avaliadas; além disso, possíveis inter-consistências entre regras de composição e de adaptação são procuradas;
3. Verificar a consistência do modelo de sistema: consiste em atestar se as duas fases anteriores foram avaliadas sem falhas e observar a presença de anomalias *"falso opcional"* e *"característica morta"* e verificar se ao menos um produto pode ser derivado;
4. Verificar o produto configurado: verifica se o produto configurado obedece às regras de composição bem como às regras de boa formação de um produto; e
5. Verificar os produtos adaptados: verifica se alguma adaptação pode adaptar o produto de forma que ele desobedeça uma regra de composição ou de boa formação.

O fluxograma dessas cinco fases é visto na Figura 5.7. A notação Bussines process modeling notation (BPMN) (BPMN, 2012) foi usada nessa figura. Ela exibe, de forma geral, as entradas e saídas do PRECISE. Além disso, a mesma figura mostra a interface de uso que o engenheiro de software vai ter acesso. O PRECISE checa uma LPSSC observando os meta-modelos (diagrama de características, diagrama de contexto, regras de composição e regras de contexto) em conjunto com um produto configurado pelo engenheiro de software.

A execução do PRECISE inicia quando o engenheiro de software solicita a verificação da LPSSC. Para tanto, o engenheiro de software fornece a LPSSC a ser verificada. Esta solicitação faz com o que o PRECISE inicie a fase 1 que recebe como entrada o diagrama de características e o diagrama de contexto da LPSSC. Ambos meta-modelos são avaliados quanto à corretude. Se os meta-modelos são avaliados como incorretos, o sub-processo *"Analisar resultado"* é invocado e o engenheiro de software é informado dos erros detectados. Em caso de sucesso, o PRECISE avança para a fase 2.

A fase 2 do PRECISE recebe como entrada as regras de composição e de contexto bem como os diagramas de característica e de contexto. Nesta fase, o sub-processo *"Verificar regras"* é aplicado para as regras de composição e para as regras de contexto. Da mesma forma que ocorreu na fase 1, em caso de falha na verificação, o subprocesso *"Analisar resultado"* é invocado e o engenheiro de software é informado dos erros. Durante todo o PRECISE, esse sub-processo é invocado quando ocorre a falha em alguma das fases.

A fase 3 recebe como entrada o diagrama de características e as regras de composição. O sub-processo *"Verificar Consistência"* consiste de verificar se é possível derivar, pelo menos, um produto da LPSSC.

A fase 4 verifica a corretude do produto configurado pelo engenheiro de software. Essa verificação corresponde ao sub-processo *"Verificar PD"* e a entrada consiste do produto configurado, do diagrama de características e das regras de composição.

Por fim, a fase 5 verifica em tempo de desenvolvimento a corretude dos produtos após sofrerem adaptações devido a mudanças contextuais. Essa verificação é realizada através de uma simulação onde a entrada de dados consiste dos meta-modelos diagrama de características e de contexto, regras de composição e de contexto e do produto configurado na fase 4.

A simulação invoca os processos *"Simular Cenários"* e *"Verificar PA"* de forma iterativa. Dado um número máximo de iterações estipulado pelo engenheiro de software, cenários de simulação são requisitados e aplicados sobre o produto corrente que poderá sofrer uma adaptação. Essa iteração continua até que o número máximo de iterações seja atingido.

O PRECISE não estipula que todas as fases de verificação necessitem ser executadas. Porém, na FixTure essa condição é imposta com o objetivo de garantir que as fases do processo sejam executadas na ordem pré-estabelecida no PRECISE.

Uma vez que foram mostrados e explicados os meta-modelos no Capítulo 4, é interessante notar que eles não operam isoladamente. Por exemplo, a criação de um novo *snapshot* pode ativar outras regras de contexto. Por sua vez, as regras de contexto podem alterar o produto corrente que, por fim, deve ser verificado se continua obedecendo às regras de boa formação e de composição.

O PRECISE oferece um processo para detecção de inconsistências nos diagramas de características e de contexto e nas regras de composição e de adaptação.

A execução manual da implementação desse processo é cansativa e tende a ser propensa a erros. Desta forma, a ferramenta FixTure tem como um de seus objetivos propor uma automatização da execução desse processo.

5.3.1 Fase 1 - Verificação de diagramas

A fase 1 tem como objetivo verificar a boa formação dos diagramas de características e de contexto. Esta fase é implementada através de restrições em EVL. Neste sentido, todas as restrições definidas no trabalho de doutorado (MARINHO, 2012) o qual esta dissertação toma como base foram analisadas e transformadas em EVL. Algumas novas regras tiveram de ser adicionadas pelo fato de os meta-modelos (diagramas e regras) serem mais aprofundados nesta dissertação.

Ao todo, 63 regras de boa formação (incluindo fase 1, fase 2 e fase 4) foram definidas na implementação do processo. Essas 63 regras foram implantadas também na FixTure. Dada a quantidade de regras, nesta seção apenas algumas regras são exibidas, porém elas podem

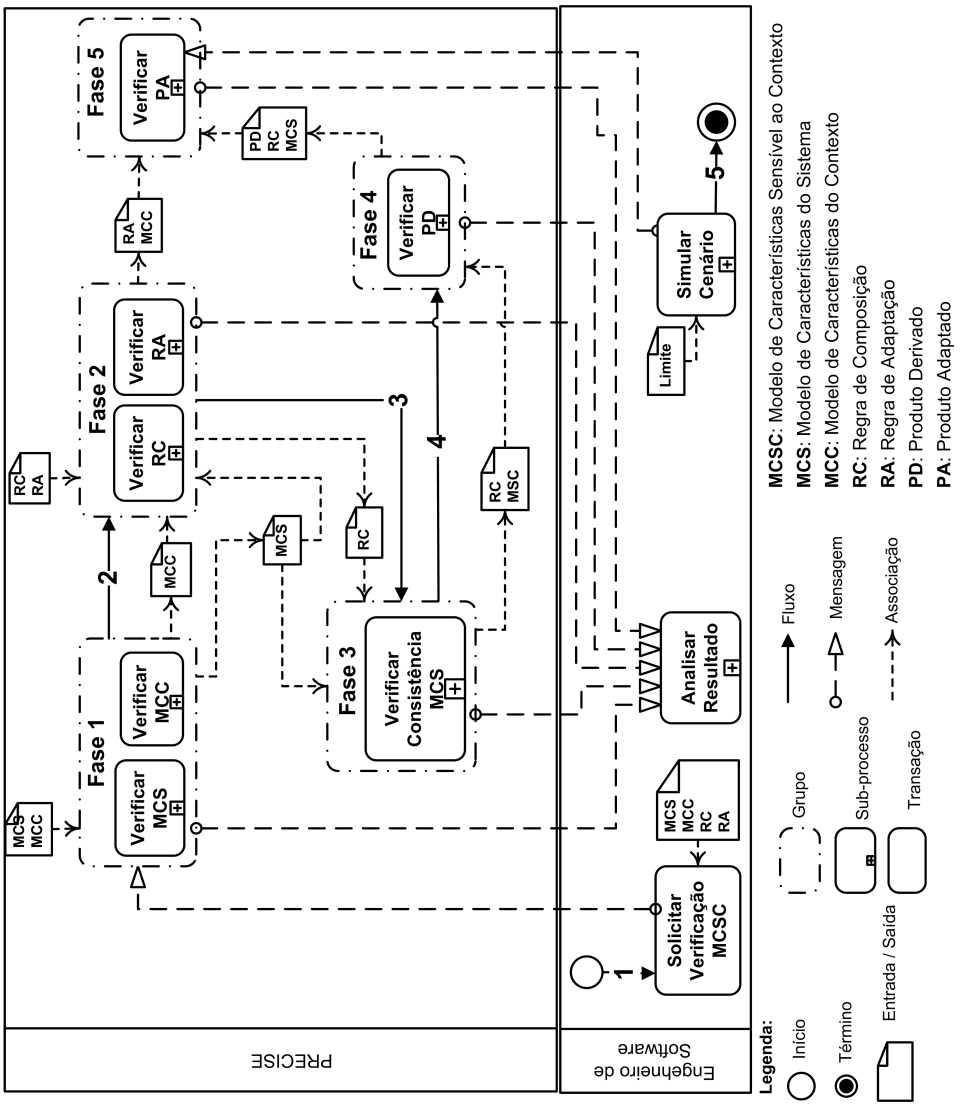


Figura 5.7: Fluxograma do PRECISE (MARINHO, 2012)

ser vistas integralmente no site do MobiLine (MOBILINE, 2012).

Na Figura 5.8, há dois exemplos de regras de boa formação que lidam com boa formação do diagrama de características os quais, juntamente, com as regras de composição constituem o modelo de sistema. Como pode ser visto, a primeira restrição *"MandatorioPrecedidoPorMandatorio"* se aplica a todos as instâncias da classe *"CaracteristicaMandatoria"* e verifica se o pai da característica mandatória é outra característica mandatória ou se é uma caracterísitica raiz. Essa restrição em EVL é destacada na Listagem 5.3.1.

```
1  CHECK(self.caracteristicaPai.isTypeOf(CaracteristicaMandatoria)
2    or self.caracteristicaPai.isTypeOf(CaracteristicaRaiz) , "fase 1")
```
(5.1)

A operação *"CHECK"* recebe dois parâmetros: um do tipo *"booleano"* e um do

```
144 context CaracteristicaMandatoria
145 {
146     //RBF 9
147     constraint MandatorioPrecedidoPorMandatorio
148     {
149         guard: self.caracteristicaPai<>null
150         check: CHECK(self.caracteristicaPai.isTypeOf(CaracteristicaMandatoria)
151                 or self.caracteristicaPai.isTypeOf(CaracteristicaRaiz) , "fase 1")
152
153         message: 'Um elemento mandatório deve sempre ser precedido por um elemento mandatório'
154     }
155 }
156
157 context CaracteristicaOpcional
158 {
159     //RBF 10 pt1
160     constraint OpcionalEspecializaPorOpcional
161     {
162         guard: self.caracteristicaFilha.size()<>0
163         check: CHECK(not self.caracteristicaFilha->
164                             exists(x|x.isTypeOf(CaracteristicaMandatoria)) , "fase 1")
165
166         message: 'Um elemento opcional napo pode ser especializado por um elemento mandatorio'
167     }
168 }
```

Figura 5.8: Exemplo de regra em EVL

tipo *"string"*. Esta operação auxiliar guarda o resultado booleano na fase correspondente ao segundo parâmetro. No exemplo mostrado na Listagem , ele guarda a verificação na *"fase 1"*.

A segunda verificação, exposta na Listagem 5.2, se aplica a todas as instâncias de *"CaracteristicaOpcional"* e verifica se a instância não possui filhos do tipo *"Caracteristica-Mandatoria"*.

```
1  check: CHECK(not self.caracteristicaFilha->
2    exists(x|x.isTypeOf(CaracteristicaMandatoria)) , "fase 1")
```
(5.2)

```
461 context EntidadedeContexto
462 {
463     constraint ElementodeContextoSemAtributos
464     {
465         check: CHECK(self.informacoesdeContexto.size() > 0, "fase 1")
466
467         message: 'O elemento de contexto deve estar associado a pelo menos um atributo de contexto'
468     }
469 }
470
471 context InformacaodeContexto
472 {
473     constraint AtributodeContextoSemPai
474     {
475         check: CHECK(self.elementoPai <> null, "fase 1")
476
477         message: 'O atributo de contexto deve estar associado a algum elemento de contexto'
478     }
```

Figura 5.9: Regra em EVL para verificação do diagrama de contexto

A fase 1 também trata os elementos do diagrama de contexto. A Figura 5.9 mostra duas regras de boa formação sobre elementos de contexto. A primeira regra da Figura 5.9 estipula que todas as instâncias da classe *"EntidadedeContexto"* têm que ter pelo menos uma

entidade da classe *"InformacaodeContexto"* como filha. Essa checagem é feita em EVL e exibida na Listagem 5.3.

```
1  check: CHECK(self.informacoesdeContexto.size() > 0, "fase 1")
```
(5.3)

A segunda regra de boa formação da Figura 5.9 se aplica a todas as instâncias da classe *"InformacaodeContexto"* e exige que todas tenham um elemento da classe *"EntidadedeContexto"* como pai. O código em EVL é exibido na Listagem 5.4.

```
1  check: CHECK(self.elementoPai <> null, "fase 1")
```
(5.4)

5.3.2 Fase 2 - Verificação de regras

```
229 context RegraDeComposicao
230 {
231     constraint RegradeComposicaoAntecedente
232     {
233         check: CHECK(self.antecedente <> null, "fase 2")
234 
235         message: 'A regra de composicao necessita de um antecedente'
236     }
237     constraint RegradeComposicaoConsequente
238     {
239         check: CHECK(self.consequente <> null, "fase 2")
240 
241         message: 'A regra de composicao necessita de um consequente'
242     }
243 }
244 context RegraDeContexto
245 {
246     constraint RegradeContextoEvento
247     {
248         check: CHECK(self.evento <> null, "fase 2")
249 
250         message: 'A regra de contexto necessita de um evento'
251     }
252     constraint RegradeContextoAcao
253     {
254         check: CHECK(self.acao <> null, "fase 2")
255 
256         message: 'A regra de contexto necessita de um acao'
257     }
258 }
```

Figura 5.10: Regras de boa formação sobre as regras de composição e de adaptação

A fase 2 do PRECISE consiste da verificação da boa formação das regras de composição e de adaptação. Além disso, também faz parte da fase 2, verificar se há inconsistências entre as regras de composição e entre as regras de contexto.

A primeira parte é implementada, semelhantemente à fase 1, com o uso de restrições em EVL. Na Figura 5.10 é possível ver duas restrições em relação às regras de composição e duas restrições em relação às regras de contexto.

As duas regras de boa formação em relação às regras de composição estipulam que qualquer instância de *"RegraDeComposicao"* deve ter as propriedades *"antecedente"* e *"consequente"* não nulas. Em relação às regras de contexto, as duas regras de boa formação

estipulam que qualquer instância de *"RegraDeContexto"* deve ter as propriedades *"evento"* e *"ação"* não nulas.

A segunda parte da fase 2 consiste em ver as inconsistências nas regras de composição e nas regras de contexto.

Afirma-se que as regras de composição contêm inconsistências entre si, quando a conjunção total das regras de composição não é satisfazível. Para avaliar se a conjunção total das regras de composição é satisfazível ou não, nesse trabalho usa-se um DDB. Para que o uso do DDB seja possível é necessário que os modelos desenvolvidos nesse trabalho sejam transformados. Essa transformação consiste em criar a partir das classes, fórmulas proposicionais e submete-las ao DDB. É interessante observar que essa transformação serve não só para a fase 2 mas também para a fase 4 e fase 5. Esse processo consiste em transformar o diagrama de características e as regras de composição em fórmulas booleanas. Este processo é detalhado na Seção 5.2.

O primeiro tipo de inconsistência nas regras de contexto ocorre quando duas regras de contexto estipulam ações contraditórias entre si. Por exemplo, suponha as seguintes três regras de contexto a seguir (ver Figuras 5.1 e 5.2):

- RA4: $carga\_bateria > 50 \rightarrow$ (luminosidade $\leftarrow 45$) AND (INSIRA $CAMERA\_FOTOGRAFICA$)
- RA5: $sinal\_gps > 25$ AND $carga\_bateria$ <40 $\rightarrow$ (INSIRA $REPRODUCAO\_VIDEO$) AND (REMOVA $CAMERA\_FOTOGRAFICA$)
- RA6: $distancia\_de\_casa > 100$ AND $sinal\_gps < 70 \rightarrow$ (luminosidade $\leftarrow 20$) AND (REMOVA $REPRODUCAO\_VIDEO$)

Os eventos das regras RA4 e RA6 podem ser ativados simultaneamente. Neste caso, são realizadas as seguintes ações: $(luminosidade \leftarrow 45)$ e $(luminosidade \leftarrow 20)$ AND $(REMOVAREPRODUCAO\_VIDEO))$. Estas duas regras, RA4 e RA6, estipulam duas ações $(luminosidade \leftarrow 45)$ e $(luminosidade \leftarrow 20)$ para o mesmo atributo "luminosidade". Elas são contraditórias, pois enquanto a ação da regra RA4 designa o valor "45" para luminosidade, a ação da regra RA6 designa o valor 20 para o mesmo atributo. Como não é possível designar dois valores simultaneamente para um só atributo, então essas regras são identificadas como sendo contraditórias entre si.

Na mesma linha de raciocínio, as regras RA4 e RA5 possuem ações contraditórias: "(luminosidade $\leftarrow$ 45) AND (INSIRA $CAMERA\_FOTOGRAFICA$)" para RA4 e "(INSIRA $REPRODUCAO\_VIDEO$) AND (REMOVA $CAMERA\_FOTOGRAFICA$) para RA5). Enquanto a regra RA4 indica a inserção da característica opcional $CAMERA\_FOTOGRAFICA$, a regra RA5 assinala a remoção desta mesma característica. Porém, é necessário observar que estas duas regras jamais podem ser ativadas simultaneamente por conta dos seus "eventos". Assim, é necessário um algoritmo para identificação de regras de contexto disparadas simultaneamente que contém ações contraditórias entre si. Portanto, é necessário descobrir as regras de contexto que são concorrentes (podem ser ativadas simultaneamente) e contraditórias (entre si).

O Algoritmo 2 consiste em primeiramente descobrir quais regras são ativadas simultaneamente e quais regras são contraditórias entre si. O funcionamento do algoritmo contém três passos.

Algoritmo 2 Algoritmo para detecção de regras de contexto concorrentes e contraditórias

```
1: Entrada: regras de contexto RC
2: Declare: pares
3: Declare: snapshots
4: Declare: disjuncao
5: for all rc_i in RC do
6:    Dis ← Dis ∨ rc_i.evento
7: end for
8: snapshots ← BDD.Solve(Dis)
9: for all snap_i in snapshots do
10:   for all rc_i in RC do
11:      if BDD.Solve(RC_i.evento, snap_i) then
12:         pares[snap_i].add(rc_i.acao)
13:      end if
14:   end for
15:   if pares[snap_i] contém ações inconsistentes then
16:      Informe ao usuário.
17:   end if
18: end for
```

O ***primeiro passo*** consiste em descobrir como satisfazer a disjunção das regras de contexto. Sabendo quais os valores das informações de contexto satisfazem pelo menos uma regra de contexto, submete-se cada um dos valores e verificam-se quais regras são ativadas simultaneamente. Neste ponto, faz-se necessário questionar se o mais simples não seria verificar todos os possíveis pares de regras (supondo que haja R regras de contexto, haveria $(R^2 - R)/2$ pares de regras) e verificar quais delas, em caso de serem ativadas simultaneamente, são contraditórias entre si. Embora pareça mais simples essa abordagem, é necessário frisar que para a fase de simulação é preciso saber quais as regras de contexto que podem ser ativadas simultaneamente. O procedimento de verificar par-a-par as regras de contexto checa apenas as ativações de duas regras de contexto, porém nada impede que haja ativação de três, quatro ou mais regras de contexto simultaneamente.

O algoritmo para descobrir quais regras são ativadas simultaneamente consiste em realizar uma disjunção ("OU") dos eventos de cada regra de contexto. Para que seja possível submeter a disjunção do evento de cada uma das regras de contexto, é necessário transformar um evento em uma fórmula proposicional. Dado que o evento de uma regra de contexto consiste de conjunções e disjunções de instâncias de *"InformacaoDeContexto"*, basta que estas instâncias sejam transformadas em variáveis booleanas. O procedimento para transformar uma instância de *"InformacaoDeContexto"* em uma variável booleana é similar ao realizado para transformar atributos de uma característica em uma variável booleana. É possível adaptar o procedimento de transformação de atributos em variáveis booleanas para o procedimento de transformar instâncias *"InformacaoDeContexto"* em variáveis booleanas. Isto se deve ao fato

que é possível que várias instâncias de *"EventoRelacional"* façam referência a mesma instância de *"InformacaoDeContexto"*.

Verifique estas três regras de contexto, RA_1, RA_2 e RA_3 (ver Figuras 5.1 e 5.2). As ações dessas três regras de contexto foram suprimidas, pois, para o que se deseja discutir no momento, são irrelevantes).

- RA_1 gps_sinal > 32 $\leftarrow \ldots$
- RA_2 distancia_de_casa > 41 $\leftarrow \ldots$
- RA_3 gps_sinal > 50 AND distancia_de_casa < 20 $\leftarrow \ldots$

Transformando as instâncias de "InformacaoDeContexto" para variáveis booleanas, têm-se:

- gps_sinal > 32 : V_{ic1}
- distancia_de_casa > 41 : V_{ic2}
- gps_sinal > 50 : V_{ic3}
- distancia_de_casa < 20 : V_{ic4}

Logo, as regras de contexto RA_1, RA_2 e RA_3, são agora fórmulas proposicionais:

- RA_1 $V_{ic1} \leftarrow \ldots$
- RA_2 $V_{ic2} \leftarrow \ldots$
- RA_3 V_{ic3} AND $V_{ic4} \leftarrow \ldots$

Veja que as regras de contexto RA_1 e RA_2 podem ter seus eventos facilmente ativados simultaneamente, pois estes lidam com instâncias de *"InformacaoDeContexto"* totalmente desconexas (a saber, *gps_sinal* e *distancia_de_casa*). Porém, o mesmo não pode ser dito das regras de contexto RA_2 e RA_3. Em verdade, as regras de contexto RA_2 e RA_3 nunca são ativadas simultaneamente, pois os eventos dessas regras não são satisfazíveis ao mesmo tempo, pois quando *"distancia_de_casa > 41"* é verdadeiro, *"distancia_de_casa < 20"* não é e vice-versa. Neste sentido, uma vez que se deseja saber todas as possíveis ativações simultâneas de regras de contexto, é necessário que esses casos de interdependência entre eventos sejam considerados a fim de que não sejam permitidos casos como permitir a ativação simultânea das regras RA_2 e RA_3. Assim, é necessário identificar todas estas restrições.

Para calcular as interdependências entre as variáveis booleanas criadas para instância de "InformacaoDeContexto", emprega-se o mesmo algoritmo 3 usado para detectar as interdependências entre as variáveis booleanas criadas para cada instância de *"ExpressaoRelacional"* (usado nas regras de composição).

Ao submeter essa fórmula proposicional ao DDB são obtidos todos os *"snapshots"* (Ver Seção 4.2.3) que satisfazem pelo menos uma regra de contexto. Cada *snapshot* ativa uma ou mais regras de contexto. Com essa informação em posse, são verificadas quais regras de contexto são ativadas pelo *snapshot* avaliado.

O funcionamento dessa verificação é detalhada no Algoritmo 2.

As linhas 1, 2, 3 e 4 criam as variáveis auxiliares usadas ao longo do Algoritmo 2: *RC* é usada como entrada e armazena as regras de contexto, *pares* é usada como estrutura *chave-valor*, onde a chave é um conjunto de valores de *"InformacaoDeContexto"*, anteriormente, chamado de *"snapshot"*, e valores é um array de regras de contexto significando que um certo *"snapshot"* ativou simultaneamente algumas regras de contexto), *"snapshots"* que armazena as formas de satisfazer a disjunção criada nas linhas 5, 6 e 7 e *"disjuncao"* é a disjunção total de todas as regras de contexto.

As linhas 5, 6 e 7 criam uma disjunção com toda as regras de contexto. O objetivo é submeter essa disjunção total ao DDB de tal forma que este identifique todas as formas de satisfazer, pelo menos, uma regra de contexto por vez. Se o DDB descobre uma forma de satisfazer essa disjunção, isso significa que há uma forma de ativar pelo menos uma regra de contexto.

A linha 9 inicia um laço que percorre todos os *"snapshots"* descobertos pelo DDB. Dentro desse laço, há um laço iniciado na linha 10 que itera sobre todas as regras de contexto. A linha 11 verifica se o *"snapshot"* $snap_i$ satisfaz a regra de contexto rc_i: se satisfazer, então o *array* de regras ativadas por $snap_i$ é acrescido da regra rc_i. A linha 15 verifica se as regras ativadas simultaneamente pelo *"snapshot"* $snap_i$ contém inconsistências entre si.

O ***segundo passo*** consiste em avaliar cada snapshot e determinar quais regras eles ativam. Cada *"snapshot"* é convertido em um conjunto de valores booleanos e então repassados para as variáveis booleanas criadas na Seção 5.2. Estes *"snapshots"* são submetidos conjuntamente ao DDB. Este, por sua vez, verifica quais regras são disparadas por quais *"snapshots"*. Assim, é criado um relacionamento *"chave-valor"* de *"snapshot"* para *"regras ativadas simultaneamente"*.

O ***terceiro passo*** consiste em analisar se as regras de contexto que podem ser ativadas simultaneamente contém ações contraditórias entre si. Uma vez que são conhecidas quais regras são ativadas simultaneamente, é possível avaliar quais delas são contraditórias entre si. Essa segunda parte é mais simples e consiste em analisar se as regras ativadas especificam ações contraditórias. Duas ações são contraditórias se:

- uma inclui e a outra exclui a mesma característica opcional; ou
- uma atribui um valor para um atributo e a outra atribui outro valor para o mesmo atributo.

Uma vez que são conhecidas as regras de contexto concorrentes e contraditórias, o usuário da FixTure é informado de tal inconsistência.

O segundo tipo de inconsistência em regras de contexto acontece quando uma

regra de adaptação define duas ações sobre a mesma características. Por exemplo, remover e adicionar a mesma característica ou atribuir valores diferentes para o mesmo atributo são exemplos desse tipo de inconsistência.

Uma inconsistência entre as regras de composição ocorre quando a conjunção de todas as regras de composição é insatisfazível. Quando isso acontece, nenhuma configuração de produto poderá satisfazer a todas as regras de composição.

5.3.3 Fase 3 - Verificação de regras

Alguns conjuntos de regras de composição podem ser criados e definidos sobre um diagrama de característica de forma tão restritiva tal que a combinação de regras de composição e diagrama de características impede que algum produto possa ser criado de forma a respeitar as regras de composição e as regras de boa formação. Nesta fase 3, é verificado se um diagrama de características juntamente com as regras de composição permitem a geração de algum produto.

Para tanto, para cada regra de composição é criada uma fórmula proposicional usando as variáveis booleanas criadas para os elementos do diagrama de característica conforme foi demonstrado na Seção 5.2. Essas regras de composição são unidas em uma conjunção (*"E"*). A essa conjunção, juntam-se as restrições entre as variáveis booleanas criadas para representar os atributos. Por fim, fechando a conjunção, concatenam-se relacionamentos transformados em fórmulas proposicionais (conforme Tabela 5.4). Para facilitar a leitura, nesta seção, vamos chamar essa conjunção de T.

Essa conjunção têm o seguinte significado quando submetido a um DDB: *é possível existir um produto que satisfaça ao mesmo tempo as regras de composição, as regras de boa formação e aos relacionamentos entre si (mandatório, opcional, alternativo "OU" e "Ou Exclusivo")*? Se existe, então o DDB vai emitir pelo menos um conjunto de valores para as variáveis booleanas indicando que a LPSSC contém pelo menos um produto.

Uma vez formulada a conjunção lógica T, é possível fazer os seguintes procedimentos para a descoberta das anomalias listadas em (BENAVIDES; SEGURA; RUIZ-CORTÉS, 2010): falso opcional e característica morta. Uma característica opcional é dita *"falso opcional"* quando não é possível criar um produto em que esta característica esteja ausente: em qualquer produto, criado esta característica (que é, *a priori*, opcional) está presente. Uma característica opcional é dita *"característica morta"*, quando não é possível criar um produto em que esta característica opcional esteja presente.

Para detectar um falso opcional para uma característica opcional o, a conjunção lógica T é submetida ao DDB passando como certificado um único valor booleano *"falso"* representando o valor da variável booleana que representa a característica opcional o. Se o DDB informa que não é possível satisfazer a conjunção T quando a variável booleana representando a característica opcional o é falso, então pode-se afirmar que a característica opcional o é um *"falso opcional"*.

Para detectar se uma característica opcional o é uma anomalia do tipo *"caracterís-*

tica morta", de modo semelhante é submetida ao DDB a conjunção lógica T passando como certificado um único valor booleano *"verdadeiro"*, indicando o valor da variável booleana que representa a característica opcional o. Se o DDB informa que não é possível satisfazer a conjunção T quando a variável booleana representando a característica opcional o tem o valor booleano *"Verdadeiro"*, então pode-se afirmar que não existe um só produto nesta LPSSC que esta característica opcional o apareça.

5.3.4 Fase 4 - Verificação de produto configurado

A fase 4 verifica se o produto configurado obedece às regras de boa formação de um produto e se obedece às regras de composição. Para o primeiro tipo de verificação, as especificações em EVL são analisadas pela ferramenta FixTure. Na Figura 5.11, é mostrado um exemplo de verificação desta fase.

```
487 context AtributoProduto
488 {
489     constraint AtributoProdutoDeveTerValor
490     {
491         check: CHECK(self.valor <> null, "fase 4")
492 
493         message: 'Deve ser atribuido algum valor ao atributo'
494     }
495 }
496 
```

Figura 5.11: Especificação em EVL para a quarta fase

Na Figura 5.11, está especificado que as instâncias de "Atributo" pertencente a um produto devem possuir algum valor. O conjunto de todas as especificações está exposto integralmente no site do MobiLine (MOBILINE, 2012).

O DDB é usado novamente para verificar se um produto P respeita todas as regras de composição. Conforme fora dito na Seção 5.2, os elementos do diagrama de características foram transformados em variáveis *boolenas* e as regras de composição foram transformadas em fórmulas proposicionais usando essas variáveis booleanas. Assim, para verificar se o produto P atende as regras de composição, o produto P será transformado em um certificado que é um conjunto de valores booleanos para as variáveis booleanas criadas. Uma vez tendo feito isso, submete-se esse certificado ao DDB para que este confirme se o certificado satisfaz a fórmula proposicional criada a partir da regra de composição. Se o certificado satisfaz à esta fórmula proposicional, então o produto não desobedece à regra de composição.

5.3.5 Fase 5 - Simulação

As fases 1, 2, 3 e 4 do PRECISE examinaram os elementos da LPSSC no intuito de eliminar o maior número possível de inconsistência e má-formação sem ter de configurar produto algum à exceção do produto verificado na fase 4. Neste sentido, o PRECISE é correto pois em uma LPSSC com *"n"* características opcionais, podemos ter até $\mathcal{O}(2^n)$ produtos possíveis. Portanto, derivar cada um deles para realizar as verificações feitas nas fases 1, 2, 3 e 4 seria

proibitivo. Contudo, algumas falhas podem permanecer latentes em uma LPSSC mesmo após as fases de 1 a 4.

Considere hipoteticamente que um diagrama de características tem as seguintes características opcionais: *"a"*, *"b"*, *"c"* e *"d"*. Considere também que uma regra de contexto *RA* estipula, por meio das ações da regra, que a característica opcional *"a"* deve ser introduzida no produto corrente ao passo que a características opcional *"c"* deve ser removida. Por fim, considere a regra de composição RC_1 e RC_2 que dizem: RC_1*: "*$a \vee b \rightarrow c$*"* e RC_1*: "*$a \wedge b \rightarrow c$*"*.

Uma vez que a regra de contexto RA é ativada, a característica opcional *"a"* é inserida no produto corrente e a característica opcional *"c"* é removida. A regra de composição RC_1 é avaliada como *violada*, pois o antecedente da regra de composição é avaliado como verdadeiro enquanto que o consequente, como falso. Pode-se afirmar que a regra de contexto *RA* sempre gera produtos que violam a regra de composição RC_1. Este caso é coberto na fase "2" do PRECISE.

Todavia, a relação da regra de contexto *RA* com a regra de composição RC_2 é inconclusiva. Analisando os antecedentes e consequentes de ambas as regras de composição RC_1 e RC_2, o antecedente da regra de composição RC_1 contém uma disjunção. Como a regra de contexto *RA* afirma, em sua ação que a característica opcional *"a"* é inserida, essa disjunção já pode ser avaliada como verdadeira, tornando assim o antecedente verdadeiro. A mesma regra de contexto *RA*, por meio de suas ações, remove a característica opcional *"c"* tornando o consequente falso.

Neste sentido, a regra de contexto *RA* estipula alterações no produto corrente que já são suficientes para afirmar que o produto corrente desobedecerá alguma regra de composição, no caso, a regra de composição RC_1. Ao dizer que as ações são suficientes para tornar o produto corrente desobediente de alguma regra de composição, pode-se entender que não é necessária avaliar a presença ou ausência de nenhuma outra característica ou valor de qualquer atributo.

Ao avaliar a regra de composição RC_2, as ações da regra de contexto *RA* são insuficientes para determinar se a adaptação tornará o produto corrente desobediente de alguma regra de composição. O motivo disso é que o antecedente da regra de composição RC_2 se baseia em uma simples conjunção (*"a"* AND *"b"*). Como a regra de contexto *RA* não estipula nenhuma ação sobre a característica opcional "b", não é possível afirmar se o antecedente será verdadeiro ou não. Como é sabido que a regra de contexto *RA* removeu a característica opcional *"c"*, tem-se o caso ilustrado na Tabela 5.5:

Tabela 5.5: Desobediência condicionada

	"a"	*"b"*	*"c"*	$\rightarrow$
RC_1	1	*"?"*	1	$1 \vee$ "?" $\rightarrow 1 : 1$
RC_2	1	*"?"*	1	$1 \wedge$ "?" $\rightarrow 1 :$ "0/1"

Dado esse cenário de falhas que só surge em face de configurações específicas, o mecanismo de simulação apresentado neste trabalho tem como objetivo enunciar o maior conjunto possível de produtos que desobedecem alguma regra de composição ou de boa formação

que não poderia ser detectado nas fases anteriores de 1 a 4 do PRECISE.

Assim sendo, neste trabalho, é simulado o ciclo de vida de uma aplicação pertencente a uma LPSSC. Por ciclo de vida, queremos dizer as diversas configurações que um produto de uma LPSSC pode assumir bem como as transições possíveis que levam de uma configuração para outra. Para tanto, usaremos o conceito de autômatos no mecanismo de simulação, denominado nesse trabalho de pesquisa de Autômato finito não-determinístico para LPSSC (AFNL).

O autômato finito não-determinístico (AFN) é uma quíntupla M definida como se segue:

$$M = \left(Q, \Sigma_\xi, \delta, q_0, F\right), \tag{5.5}$$

onde Q é o conjunto de estados que o autômato pode assumir, Σ_ξ é o conjunto de símbolos de entrada, acrescida da cadeia vazia ξ, δ é a função de transição, $q_0 \in Q$ representa o estado inicial do autômato, e $F \subseteq Q$ é o conjunto de estados finais do autômato. A função de transição δ é definida como se segue:

$$\delta = (Q \times \Sigma_\xi \rightarrow \mathscr{P}(Q)), \tag{5.6}$$

onde a entrada desta função é um par ordenado $\left(q \in Q, \sigma \in \Sigma_\xi\right)$ e cuja saída é $\mathscr{P}$. $\mathscr{P}(Q)$ é o conjunto das partes de Q.

Para usar o AFN em LPSSC, algumas adaptações precisam ser feitas:

- Em AFN, há o conjunto finito de todos os possíveis estados Q; em AFNL, há o conjunto de todas as possíveis configurações de produtos P;
- Em AFN, há o estado inicial q_0; em AFNL, há o produto configurado inicial p_0;
- Em AFN, há o conjunto de estados finais F; em AFNL, há o conjunto de produtos configurados finais F;
- Em AFN, há a função de transição δ; em AFNL também há a função de transição δ; e
- Em AFN, há o conjunto de símbolos de entrada, acrescida da cadeia vazia Σ_ξ; em AFNL, há os valores das instâncias da classe *"InformacaoDeContexto"*.

Um AFNL L é definido como se segue:

$$L = (P, \Sigma, \delta, p_0, F) \tag{5.7}$$

, onde P é o conjunto representando todos as configurações que os produtos da LPSSC pode assumir, Σ são os valores de contexto relevantes para a LPS (isto é, os valores das instâncias da classe *"InformacaoDeContexto"*), δ é a função de transição,$p_0 \in Q$ representa o produto inicial da LPS do autômato, e $F \subseteq Q$ é o conjunto de configurações de produtos inválidas.

Nesse ponto, é importante que sejam definidos a função de transição δ, o conjunto de produtos configurados finais F, e o alfabeto de entrada Σ.

Tal qual no AFN, é necessário definir um alfabeto de entrada Σ que contemple todas as entradas que o autômato pode ler. O alfabeto de entrada Σ é necessário para definir a função de transição δ.

As regras de contexto definidas para uma LPSSC servem para definir o alfabeto de entrada para o AFNL. Mais especificamente, todos os possíveis subconjuntos desse conjunto de tamanho igual ou maior a 1 ($\mathscr{P}_{\geq 1}(RA)$) são possíveis candidatos a funcionarem como entrada. A justificativa baseia-se no seguinte fato: tal qual no AFN, à medida que as entradas são processadas pelo autômato, este altera o seu estado atual. No AFNL, à medida que uma ou mais regras de contexto são ativadas simultaneamente (e não são contraditórias entre si) ocorre a alteração da configuração atual do produto mesmo que seja uma auto-adaptação, isto é, uma adaptação leva a configuração de produto para ela mesma.

Para que os subconjuntos de tamanho igual ou maior a 1 efetivamente funcionem como entrada, é necessário que os eventos das regras contidas em um subconjunto específico possam ser ativadas simultaneamente. Suponha que há 5 regras de contexto em uma LPSSC qualquer e que essas regras formam um conjunto chamado *RA*. Considere todos os subconjuntos desses conjuntos com tamanho igual ou maior a 1 (representado por $\mathscr{P}_{\geq 1}(RA)$), a conjunção de cada uma das ações de cada uma das regras de um desses subconjuntos formam um elemento do alfabeto de entrada.

Para exemplificar, considere as seguintes cinco regras de contexto:

- ra_1: $ev_1 \rightarrow ac_1$
- ra_2: $ev_2 \rightarrow ac_2$
- ra_3: $ev_3 \rightarrow ac_3$
- ra_4: $ev_4 \rightarrow ac_4$
- ra_5: $ev_5 \rightarrow ac_5$

Cada elemento $ra_1, ra_2, \ldots$; $ev_1, ev_2, \ldots$; e $ac_1, ac_2, \ldots$ são instâncias das classes *"RegraDeAdaptacao"*, *"Evento"* e *"Acao"*, respectivamente. Dado que existem cinco regras, há $(2^5 - 1)$ subconjuntos de tamanho igual ou maior a 1. São exemplos deles os subconjuntos $\{ra_1, ra_3\}$, $\{ra_4, ra_2\}$ e $\{ra_5\}$. Admita que a conjunção de ev_1 com ev_2 não é satisfazível (isto é, não há combinação de valores das instâncias das classes *"InformacaoDeContexto"* que faça ev_1 e ev_2 terem valor "Verdade" ao mesmo tempo). Consequentemente, as regras de contexto ra_1 e ra_2 não podem ser ativadas simultaneamente. Esse subconjunto $\{ra_1, ra_2\}$ não pode funcionar como entrada para o AFNL. Por outro lado, considere que a conjunção ev_1, ev_2 e ev_4 é satisfazível. Então, o subconjunto de regras de contexto $\{ra_1, ra_2$ e $ra_4\}$ pode funcionar como alfabeto de entrada.

Diferentemente da definição canônica do AFN, não consideraremos no AFN para LPSSC a cadeia vazia ξ, pois será considerado que, em um ambiente sensível ao contexto, a única forma de um produto mudar de configuração é através da ativação de uma regra de

adaptação que só é executada caso haja alguma entrada que possa ativar o lado esquerdo da regra.

A função de transição *delta* em um AFN especifica que quando certa entrada é lida pelo autômato e o mesmo se encontra em um estado Q_a, o estado do autômato é alterado para Q_b. Ainda em um AFN, as regras de transição definidas na função de transição especificam claramente que a transição ocorre de um estado específico para um ou mais outros estados.

Em um AFNL, a função de transição δ requer um alfabeto de entrada σ tal qual um AFN requer. A função de transição δ de um AFN, em geral, estipula todas as possíveis transições de um estado para outro. Tecnicamente, é necessário criar uma função de transição δ que estipule todas as possíveis transições em um AFNL de uma LPSSC. Em uma LPSSC com *"n"* características opcionais isto é uma tarefa matematicamente difícil, pois o número de possíveis produtos é $\mathcal{O}(2^n)$. Consequentemente, existem $\mathcal{O}(2^{2n})$ transições possíveis, o que impossibilita especificar todas essas transições.

Assim, o AFNL não impõe que uma transição ocorra somente de uma configuração de produto para outra específica. Na verdade, uma transição em um AFNL pode ocorrer em qualquer configuração de produto eliminando assim a necessidade de estipular todas as possíveis transições.

Portanto, a transição é disparada somente pela mudança de contexto não dependendo, desta forma, da configuração atual do produto. Logo, as transições definidas na função de transição δ são aplicáveis a qualquer configuração de produto eliminando a necessidade de especificar manualmente todas as transições possíveis, informando qual estado de partida e qual estado de chegada. .

Contudo, não é interessante permitir que uma configuração de produto que desobedece alguma regra de composição continue a se adaptar. Logo, a simulação apresentada neste trabalho considera tais tipos de configurações de produto como configurações finais, pois a partir delas não devem haver mais adaptações.

Uma configuração (válida ou não) para um produto da LPS é especificada como uma sequência de valores *"booleanos" 0* ou *1*. Cada um desses valores representa ou a pertinência de um elemento à configuração de um produto ou a satisfação de uma fórmula atômica de alguma regra de composição.

Como exemplo, para a LPS apresentada na figura 5.12 a especificação das configurações dos produtos desta LPS seria um *array* com o seguintes valores *0/1, 0/1, 0/1, 0/1, 0/1, 0/1, 0/1, 0/1, 0/1*. A Figura 5.13 mostra esta codificação. Assim, possíveis produtos podem ser codificados da seguinte forma:

- Produto 1 *Mobile Device:1, Set Profile:1, Set Roadmap:1, Context Management:1, Persistency:1, Acquisition:1, Capture:1, Via External Service:1, From Sensor:1, From Memory:1* $\rightarrow$ *1,1,1,1,1,1,1,1,1,1,1,1*
- Produto 2 *Mobile Device:1, Set Profile:0, Set Roadmap:1, Context Management:1, Persistency:1, Acquisition:1, Capture:1, Via External Service:1, From Sensor:1, From Me-*

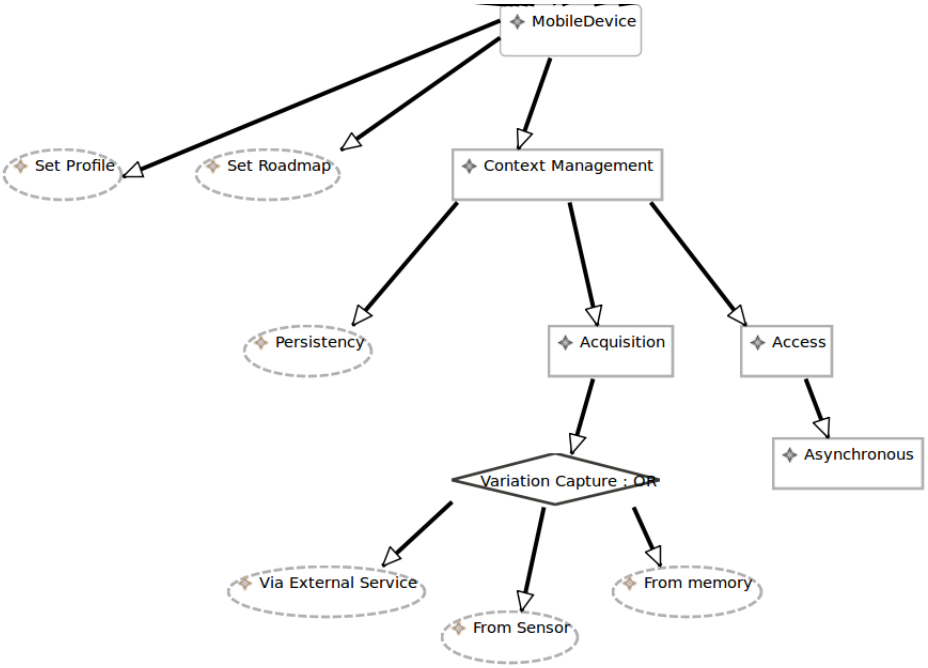


Figura 5.12: Simples diagrama de características

mory:1 → 1,0,1,1,1,1,1,1,1,1,1,1,0

- Produto 3 *Mobile Device:1, Set Profile:1, Set Roadmap:1, Context Management:1, Persistency:1, Acquisition:1, Capture:1, Via External Service:1, From Sensor:0, From Memory:1 → 1,1,1,1,1,1,1,1,1,1,0,1*

O conjunto de produtos configurados finais F contém aquelas representações (sequências de 0 e 1) que desobedecem alguma regra de composição. O motivo desta escolha é para garantir que durante a simulação uma vez que uma configuração de produto desobedece uma regra de composição não deve haver mais transições a partir dela.

O resultado da simulação é um grafo onde os vértices são considerados as configurações de produtos. Inicialmente, uma configuração de um produto é escolhida pelo usuário através da interface da ferramenta FixTure e o grafo correspondente é criado a partir dessa configuração. As arestas representam as adaptações que levam uma configuração de estado a outra.

Assim, o objetivo da simulação é identificar o maior número de vértices e arestas deste grafo, limitados por uma quantidade máxima de simulações de produtos que é informada pelo usuário. A existência desse limite superior é necessária diante da magnitude do conjunto de todos as possíveis configurações de produtos.

Para executar a simulação, foram desenvolvidas duas maneiras para o processamento do autômato definido nesta seção 5.3.5. O processamento consiste em receber como entrada um produto inicial e uma sequência de *"snapshots"*, de forma a estimular a ativação das regras de contexto, e produzir como saída o referido grafo que mostra o ciclo de vida de uma aplicação pertencente a uma LPSSC. A primeira maneira tenta percorrer tal grafo em profundidade, enquanto que a segunda tenta em largura.

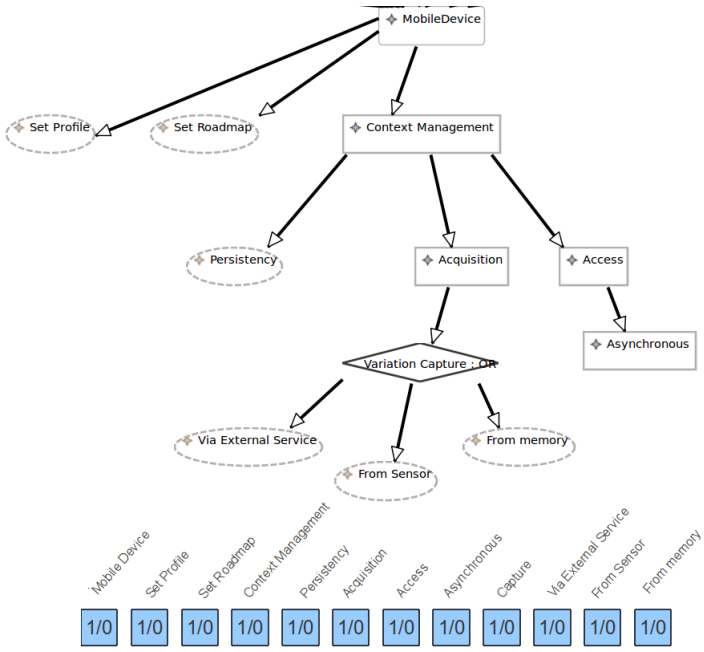


Figura 5.13: Codificação de uma LPS

5.3.5.1 Processamento em profundidade

Dado o AFNL descrito anteriormente na Seção 5.3.5, um subconjunto das regras de contexto R, chamado R^*, é escolhido e aplicado sobre um produto inicial P_0. Essa aplicação gera o produto adaptado P_1 e este é tido como produto corrente. Se o produto P_1 é reconhecido como inválido (isto é, se desobedece alguma regra de composição), essa adaptação é dita insegura e a ferramenta FixTure reporta a adaptação como insegura. Automaticamente, o produto corrente volta a ser o produto P_0. Se P_1 é válido, então a adaptação é tida como segura e P_1 é tido como o produto corrente. A condição de parada para esta forma de processamento é a quantidade máxima de adaptações (tanto seguras como inseguras) informada pelo usuário através da interface da FixTure. Esta abordagem é descrita pelo algoritmo 3.

As linhas de 1 a 7 declaram variáveis auxiliares usadas ao longo do algoritmo 3: automato *"A"*, o produto inicial p_0, a quantidade máxima de transições *"qtd"*, a quantidade máxima de transições seguras *"qtdS"*, os contadores *"t"* (para somar todas as transições realizadas) e *"ts"* (para somar somente as transições realizadas consideradas seguras) e o produto corrente que é inicializado com o produto informado p_0 pelo usuário. O laço iniciado na linha 9 diz que enquanto o número total de transições ou o número de transições seguras não ultrapassarem os seus respectivos limites, será sorteada uma subconjunto *ra* de regras de contexto com cardinalidade maior que 1 (linha 9) o que indica que uma ou mais regras de contexto foram ativadas (considerando que elas não são inconsistentes entre si). Na linha 10, as regras contidas em *ra* são aplicadas no produto corrente C gerando assim o produto c'. Se c' é um produto válido (linha 11), o produto corrente C recebe o valor de c' e os contadores de transições totais e transições seguras são incrementados e tal transição $(C \rightarrow c')$ é reportada como segura. Caso contrário, a transição $(C \rightarrow c')$ é reportada como insegura e o produto corrente C não é atualizado e apenas o contador de transições totais é incrementado.

Algoritmo 3 Algoritmo para Comportamento 1

```
1: Entrada: automato A
2: Entrada: produto inicial P_0
3: Entrada: quantidade máxima de transições qtd
4: Entrada: quantidade máxima de transições seguras qtdS
5: transições t ← 0
6: transições seguras tS ← 0
7: produto corrente C ← P_0
8: while t < qtd OU t < qtdS do
9:    Subconjunto não vazio de regras de contexto ra ⊆ 𝒫_{≥1}(RA)
10:   c' ← ra aplicado em C
11:   if c' é válido then
12:      c ← c'
13:      a transição {c → c', ra} é assinalada como segura.
14:      t ← t + 1
15:      tS ← tS + 1
16:   else
17:      a transição {c → c', ra} é assinalada como insesegura.
18:      t ← t + 1
19:   end if
20: end while
```

Para a verificação na linha 11, o produto c' é transformado em um conjunto de valores para as variáveis booleanas criadas no procedimento de transformar os meta-modelos em fórmulas booleanas para então serem submetidas ao DDB, a fim de verificar se tal conjunto (certificado) satisfaz os meta-modelos.

A Figura 5.15 mostra o funcionamento do comportamento em profundidade para a LPSSC da figura 5.14.

5.3.5.2 Abordagem em largura

Na abordagem em largura, o autômato tem um comportamento diferenciado cujo objetivo é visitar mais estados. Esse comportamento é descrito pelo Algoritmo 4. Dado o conjunto de regras de contexto R^*, partindo de um produto inicial P_0 serão executadas as adaptações referentes a cada elemento do conjunto $R*$. Dado que o conjunto $R*$ contém as regras de contexto *RA* mais as combinações de regras de contexto que podem ser ativadas simultaneamente e não são contraditórias entre si ($\mathscr{P}_{\geq>\infty}(RA)$), cada elemento de R^* é constituído ou de uma regra de contexto ou de um conjunto de regras de contexto que podem ser ativadas simultaneamente e não são contraditórias entre si.

As linhas de 1 a 6 declaram variáveis auxiliares usadas ao longo do algoritmo 4: automato *"A"*, o produto inicial p_0, a quantidade máxima de transições *"qtd"*, a quantidade máxima de transições seguras *"qtdS"*, os contadores *"t"* (para somar todas as transições realizadas) e *"ts"* (para somar somente as transições realizadas consideradas seguras) e o produto corrente que é inicializado com o produto informado p_0 pelo usuário. A linha 7 cria um vetor de

Algoritmo 4 Algoritmo para Comportamento 2

```
1: Entrada: automato A
2: Entrada: produto inicial P_0
3: Entrada: quantidade máxima de transições qtd
4: Entrada: quantidade máxima de transições seguras qtdS
5: transições t ← 0
6: transições seguras tS ← 0
7: P.add(P_0)
8: while t < qtd OU tS < qtdS do
9:    for all produto p em P do
10:      for all ra ⊆ 𝒫_{≥1}(RA) do
11:         p' ← ra aplicado em produto
12:         if p' é válido then
13:            P.add(p')
14:            a transição {p → p', ra} é assinalada como segura.
15:            t ← t + 1
16:            tS ← tS + 1
17:         else
18:            a transição {p → p', ra} é assinalada como insesegura.
19:            t ← t + 1
20:         end if
21:      end for
22:   end for
23: end while
```

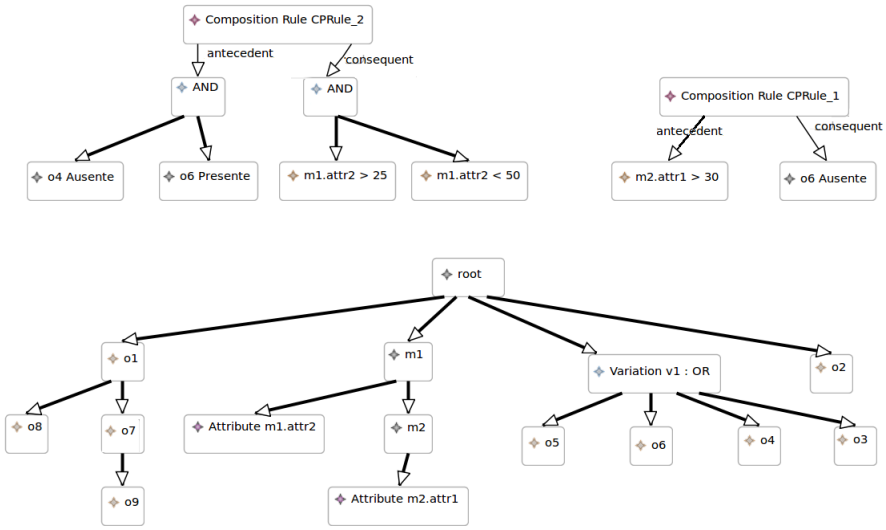


Figura 5.14: Diagrama de características para a simulação da Figura 5.15

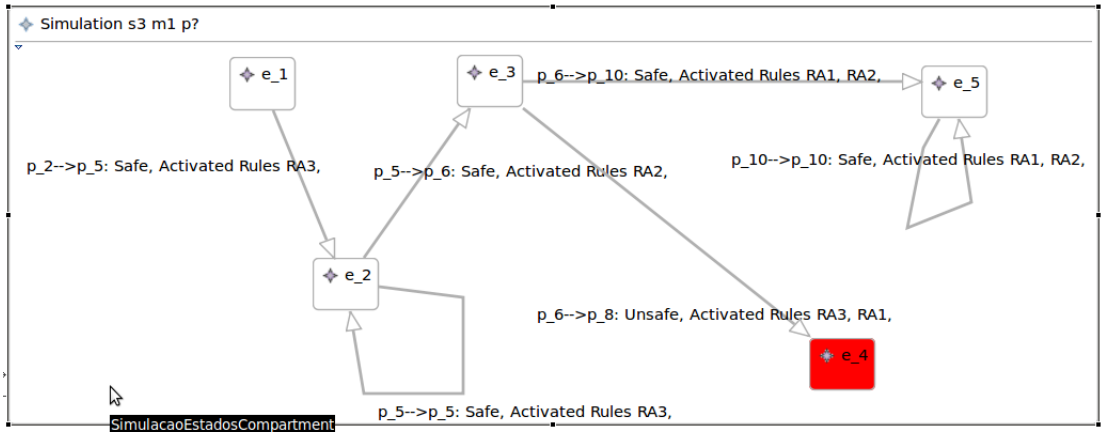


Figura 5.15: Comportamento em profundidade

produtos e o inicializa com o produto p_0. A linha 8 inicializa um laço que só deixa de executar quando pelo menos uma das duas variáveis *"t"* ou *"tS"* alcançam seus respectivos limites.

Na linha 9 do Algoritmo 4, para cada produto p em P (inicialmente, o conjunto P só contém o produto p_0) são aplicadas todas os subconjuntos de regras de contexto de tamanho maior que um e que não são inconsistentes entre si (linha 10). Essa aplicação gera um produto p' que, se for válido (linha 12), é adicionado ao conjunto P. Se p' é válido então a transição $p \rightarrow p'$ é assinalada como segura e os contadores de transições totais e o de transições seguras são incrementados. Em caso contrário (linha 17), o produto p' não é incorporado no conjunto P, a transição $p \rightarrow p'$ é reportada como insegura e apenas o contador de transições totais é incrementado. O fato de p' ser adicionado ao conjunto P indica que na próxima iteração do laço iniciado na linha 9, ele irá gerar novas transições.

Partindo de um produto inicial P_0, são executadas as adaptações referentes a cada um dos elementos de R^*. São gerados então $\mathcal{O}(|R^*|)$ novos produtos. Em seguida, para cada um desses $|R^*|$ produtos que ***não*** desobedecem nenhuma regra de composição, são geradas novamente as $|R^*|$ adaptações, gerando então agora $\mathcal{O}(|R^*| * |R^*|)$ novos produtos. Para os produtos

Figura 5.16: Comportamento em largura

gerados que desobedecem alguma regra de composição, não há mais adaptações partindo deles.

É importante ressaltar que a quantidade de novos produtos gerados pode ser mais alta que na abordagem em profundidade. Por este motivo, a condição de parada desta segunda abordagem é a quantidade máxima de produtos gerados que deve ser informada pelo usuário através da interface da ferramenta FixTure.

A Figura 5.16 mostra o funcionamento da abordagem em largura para a LPSSC da Figura 5.14.

5.4 Interfaces gráficas e uso da FixTure

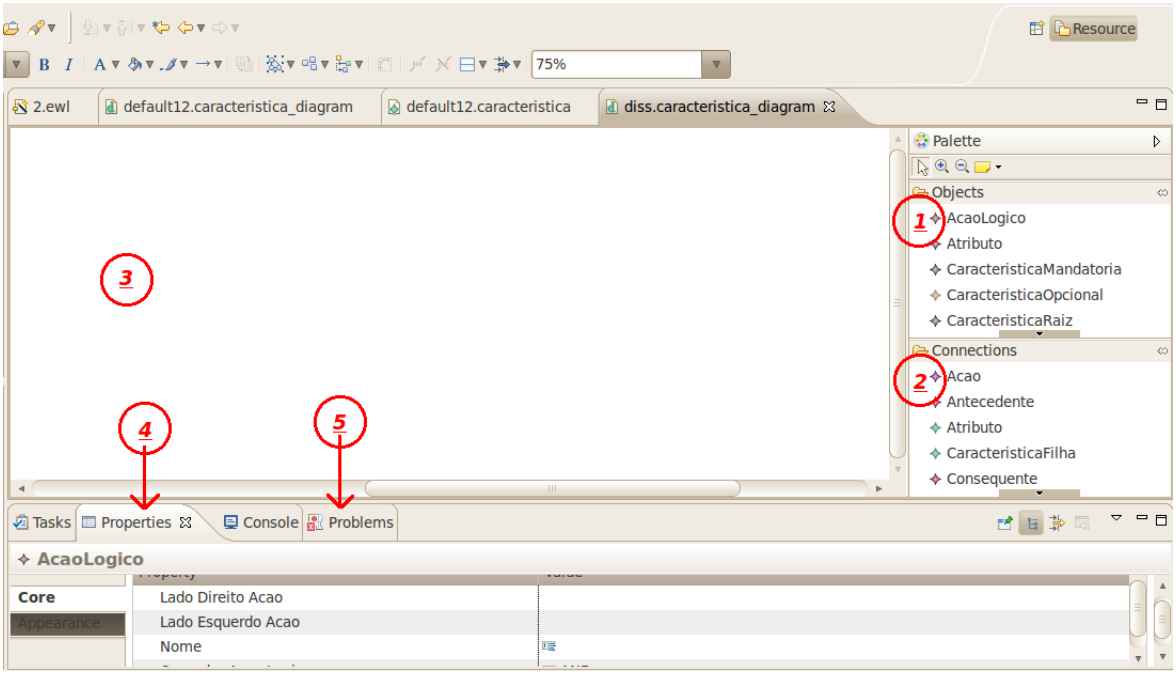


Figura 5.17: Visão inicial da ferramenta FixTure

A ferramenta FixTure foi desenvolvida como um plugin para o Eclipse (ECLIPSE, 2012). A implementação apresentada neste trabalho usou lógica proposicional e restrições escritas em EVL. Além disso, a FixTure usou a biblioteca JDD (JDD, 2012) DDB. Para gerar a parte gráfica da ferramenta, o framework Epsilon (EPSILON, 2012) foi usado.

A interface inicial da ferramenta é exibida na Figura 5.17 onde existem cinco regiões destacadas.

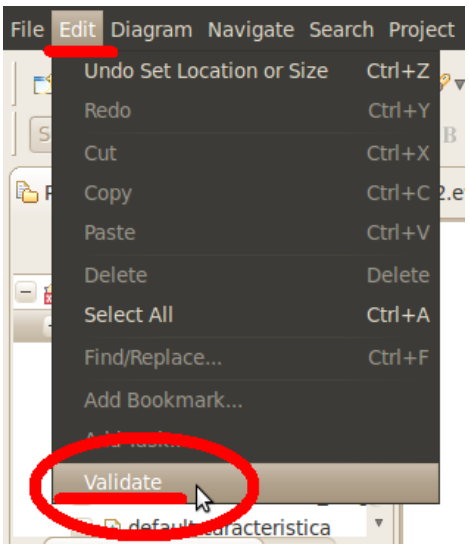


Figura 5.18: Validação inicial

A primeira e a segunda regiões denotam a palheta de objetos e conexões, respectivamente. Na primeira região, temos os objetos como *"Atributo"*, *"CaracteristicaOpcional"*, *"RegraDeComposicao"*, entre outros. Na segunda região, temos as conexões possíveis entre os objetos. Por exemplo, para um objeto *"CaracteristicaOpcional"* e um objeto *"Atributo"* é possível estabelecer a conexão *"Atributo"* para indicar que o atributo pertence àquela característica opcional.

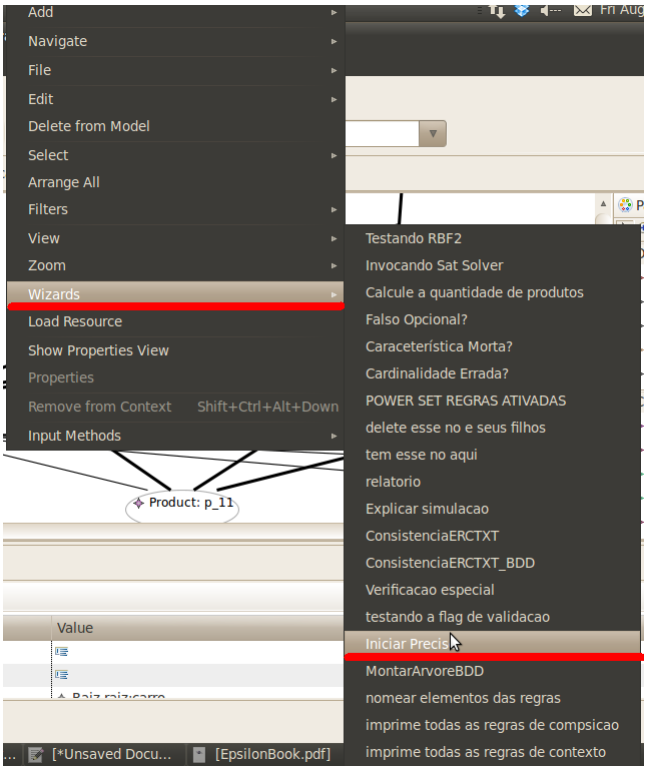


Figura 5.19: Iniciando o Precise

A terceira região da Figura 5.17 denota a área de edição. É nela onde arrastaremos os objetos e as conexões da área da palheta. A região quatro corresponde a uma aba que permite que vejamos as propriedades dos objetos e conexões que estão dispostos na terceira região. Por fim, na quinta região uma outra aba é apresentada onde é possível visualizar os problemas que tenham surgido na validação da fase 1 e fase 2 do PRECISE.

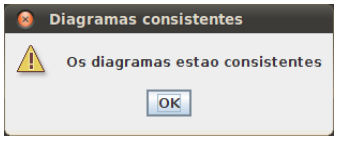


Figura 5.20: Resultado da Fase 1

Antes de iniciar o PRECISE, é preciso que o usuário da FixTure clique no Menu *"Edit"* e, em seguida na opção do Menu *"Validate"*. Essa sequência é ilustrada na Figura 5.18.

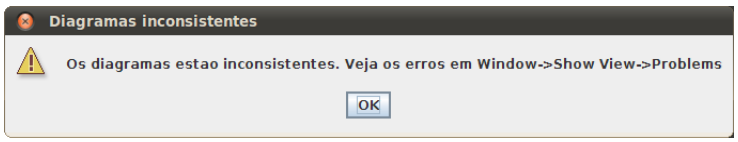


Figura 5.21: Erro na Fase 1

Para iniciar o PRECISE, deve-se clicar em uma área em branco na terceira região com o botão direito do mouse e então escolher a opção *"Wizards"* e depois em *"Iniciar PRECISE"*. Esse procedimento pode ser visto na Figura 5.19

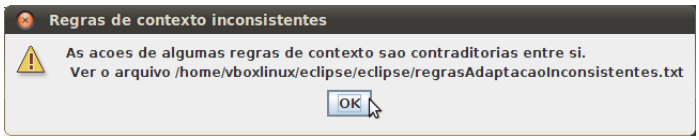


Figura 5.22: Erro na Fase 2: Regras de contexto inconsistentes entre si

Uma vez iniciado o PRECISE, as fases 1, 2, 3, 4 e 5 (simulação) são executadas em sequência. Sempre que a fase anterior tiver sido realizada com sucesso a fase posterior é iniciada. Por exemplo, se a fase 2 obtiver sucesso, então a fase 3 é executada); em caso contrário, é necessário reiniciar o PRECISE.

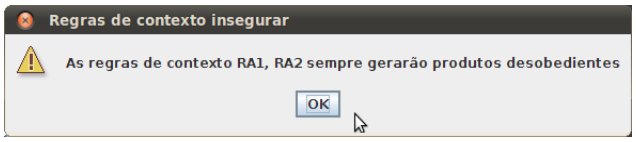


Figura 5.23: Erro na Fase 2: Regras de contexto que tornam qualquer configuração de produto desobediente a alguma regra de composição

No caso de sucesso na fase 1, a mensagem *"Os diagramas estão consistentes"* é exibida (ver Figura 5.20). Em caso de erro, a mensagem *"Os diagramas estão inconsistentes. Veja os erros em Windows->Show View->Problems"* é exibida (ver Figura 5.21) e é solicitado que o usuário consulte a região cinco da Figura 5.17, aba de problemas, da ferramenta FixTure.

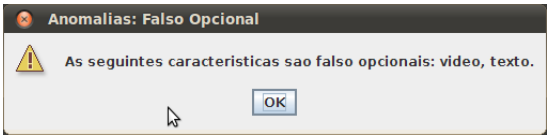


Figura 5.24: Erro na Fase 3: Detecção de anomalias do tipo *"falso opcional"* (video e texto são características falso opcionais)

Se a validação em EVL é realizada com sucesso e as regras de composição e as regras de contexto não apresentam nenhum tipo de problema, é dito que houve sucesso na fase 2 e então a fase 3 pode ser iniciada.

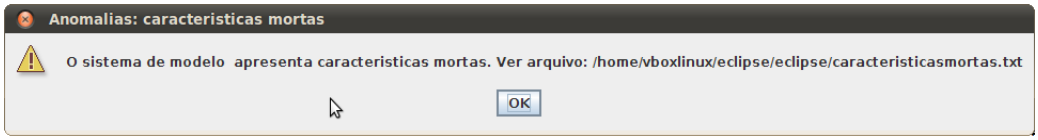


Figura 5.25: Erro na Fase 3: Detecção de anomalias do tipo *característica morta opcional"*

Se durante a fase 2, a ferramenta FixTure detecta regras de contexto inconsistentes, ela apresenta duas mensagens de erro: a primeira para as regras de contexto concorrentes e contraditórias entre si (isto é, regras de contexto que podem ser ativadas simultaneamente e que contém ações antagônicas) e a segunda para as regras de contexto que levam qualquer configuração de produto ao estado de desobediência de regras de composição. Essas duas situações são exibidas as Figuras 5.22 e 5.23.

Para a fase 3, são verificadas as presenças das anomalias de *"falso opcional"* e *"característica morta"*. Se não forem detectadas nenhuma dessas anomalias, a FixTure avança para a fase 4 avisando que não foram detectadas anomalias.

Em caso de detecção de anomalias, as seguintes mensagens são exibidas ao usuário (ver Figuras 5.24 e 5.25):

- em caso de falso opcional: *"As seguintes características são falso opcionais* $\{carac_1\}$, $\{carac_2\}$, ..., $\{carac_n\}$*"*,
- em caso de característica morta: *"O modelo de sistema apresenta características mortas: Ver em /home/vboxlinux/eclipse/ecipse/caracteristicasmortas.txt"*. É indicado o endereço físico onde há mais informações sobre essas características mortas.

Para a fase 4, é pedido ao usuário da FixTure que especifique um produto ele deseja. A FixTure irá verificar se este produto obedece às regras de composição conforme pode ser visto na Figura 5.26.

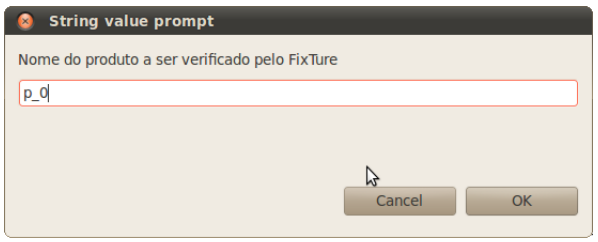


Figura 5.26: Fase 4: Produto a ser verificado em relação à obediência às regras de composição

Uma vez que o produto escolhido na fase 4 não desobedece nenhuma regra de composição, o mesmo é usado como produto inicial da simulação da quinta e última fase. Por fim, o resultado da simulação é gerado na área de edição da FixTure. Um exemplo de simulação pode ser visto na Figura 5.27.

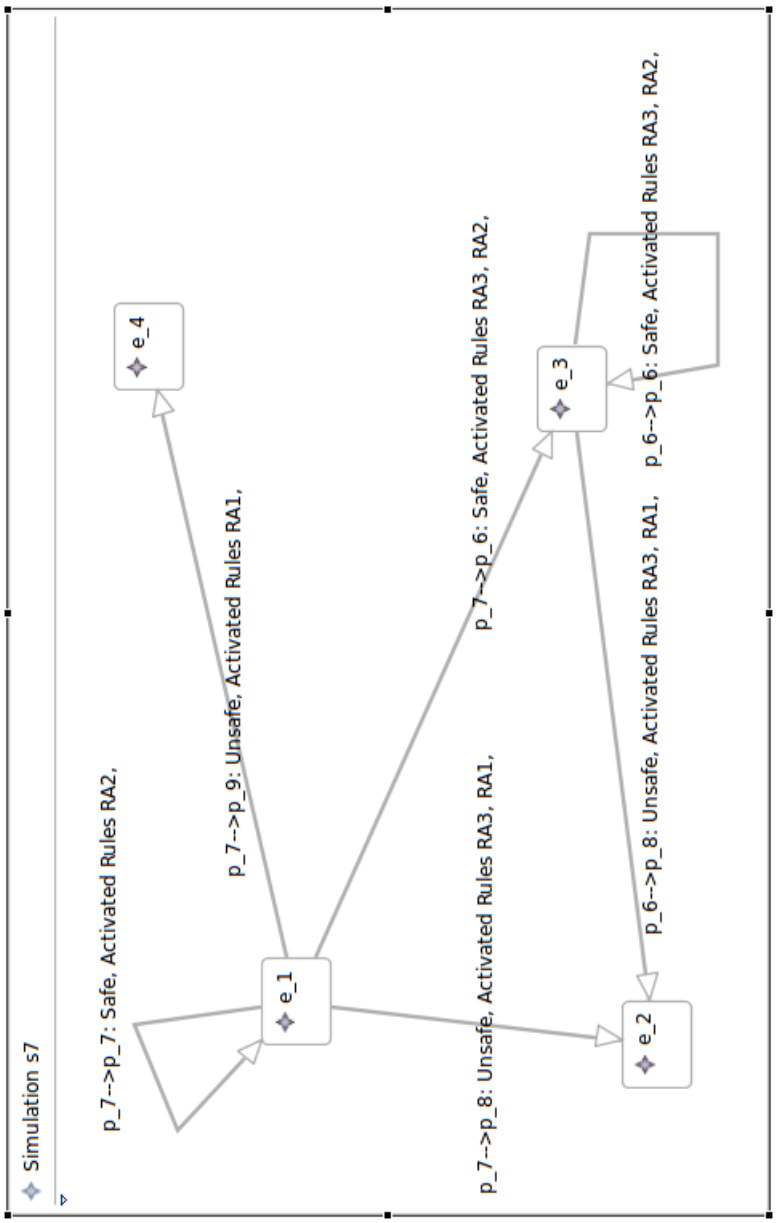


Figura 5.27: Resultado da simulação

5.5 Conclusões

Neste capítulo, foi exibida a transformação dos diagramas e das regras de composição e de contexto para fórmulas proposicionais. Esta transformação permitiu o uso de um resolvedor de fórmulas proposicionais (no caso, o DDB) para as fases 2, 3, 4 e 5 (simulação). O DDB também se mostrou eficaz na detecção de anomalias na fase 3.

Também foi tratada a questão dos atributos das características que é uma inovação da modelagem apresentada na tese de doutorado (MARINHO, 2012). Para tanto foi necessário elaborar um algoritmo para tratar a interdependência dos valores das variáveis booleanas criadas para representar um atributo.

Além disso, foram propostas duas abordagens de comportamento para a fase de simulação. Dado que o ciclo de vida do produto de uma LPSSC pode ser representado em um grafo (onde os vértices são as configurações de produtos e as arestas, as adaptações), a primeira abordagem investiga o ciclo de vida em profundidade enquanto que a segunda busca em largura. Por fim, a ferramenta FixTure foi apresentada em detalhes mostrando como o PRECISE foi implementado nela.

O próximo capítulo apresenta um estudo de caso usando o *"Guia de Visita Móveis"* que é um produto da LPSSC MobiLine (MOBILINE, 2012).

6 AVALIAÇÃO PRELIMINAR

Com o objetivo de avaliar a FixTure, neste capítulo, é desenvolvido uma avaliação preliminar usando a LPSSC *"Guia de Visitas móveis" (Mobile Visit Guide)* criada a partir da linha de produtos de LPSSC (MOBILINE, 2012). Durante a aplicação do PRECISE, foi construído um produto específico para esta dissertação chamado *"littleGreatTour"*. Na fase de simulação, cada abordagem descrita no final do Capítulo 4 foi executada 4 vezes incrementando os parâmetros que funcionavam como condições de parada da simulação.

A Seção 6.1 apresenta a LPSSC MobiLine. A partir do MobiLine é derivada uma outra LPSSC chamada *"Guia de visita móveis"*. Ainda nesta seção, os diagramas de características e de contexto e a as regras de composição e de contexto são apresentados. A Seção explica como ocorreu a avaliação preliminar. Na Seção 6.3, são apresentados alguns dos problemas detectados durante a transposição do *"Guia de visita móveis"* para a notação dos meta-modelos usados neste trabalho. A Seção 6.4 enfatiza a fase 5 da aplicação do PRECISE mostrando com detalhes a simulação. Por fim, a Seção 6.5 finaliza o capítulo tecendo comentários conclusivos sobre o foi realizado neste capítulo.

6.1 MobiLine

MobiLine é uma LPS voltada para aplicações móveis e sensíveis ao contexto. Para usar LPS para desenvolver aplicações móveis e sensíveis ao contexto, a LPS deve conter, em seu escopo, as aplicações que se encaixam em seu domínio e, através de uma diagrama de características, identificar as semelhanças e diferenças entre essas aplicações. Dado o amplo domínio, a tentativa de construir uma LPS para tal domínio de aplicações pode não ser bem sucedida. Dada essa dificuldade, a abordagem empregada no projeto do MobiLine foi dividir a análise do domínio em dois níveis: nível base e nível específico. A Figura 6.1 exemplifica essa divisão. Em ambos os níveis, há uma fase de engenharia de domínio e de engenharia de aplicação.

Com a abordagem empregada no MobiLine (MARINHO et al., 2010), o nível base contempla as funcionalidades presentes em aplicações móveis e sensíveis ao contexto. As principais funcionalidades identificadas no nível base foram ambiente de execução dinâmica, adaptabilidade e sensibilidade ao contexto.

Ao prosseguir para o nível específico, o escopo é diminuído e os requisitos passam a tratar apenas de um domínio de negócios. Um novo diagrama de características é produzido com o objetivo de abranger o domínio de negócio em conjunto com os requisitos selecionados do nível base.

O domínio de negócio do *"Guia de Visita Móveis"* é o guia de visitas móvel e sensível ao contexto. Esse domínio engloba aplicações sendo executadas em dispositivos móveis com o objetivo de ajudar visitantes que desconhecem o ambiente como museus ou parques. Abaixo do nível específico, há o produto configurado que é gerado através de uma seleção dos

componentes reusáveis identificados nos dois níveis superiores (base e específico).

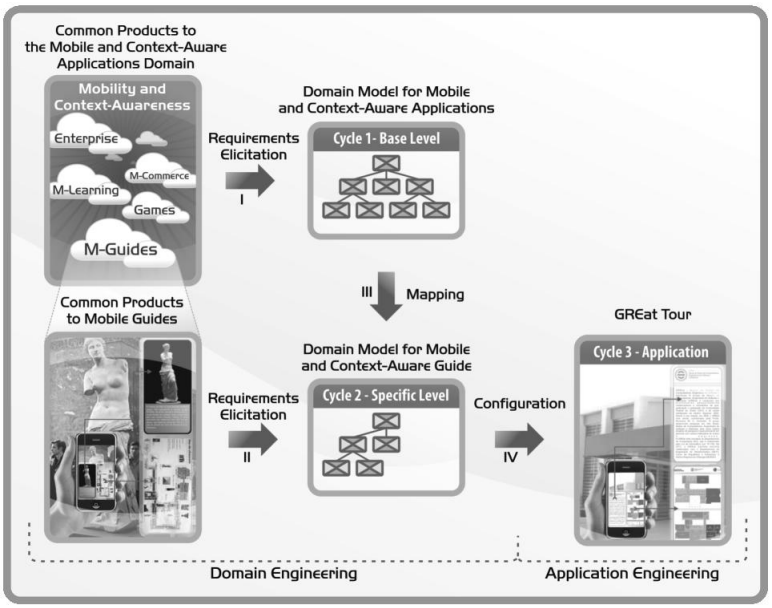


Figura 6.1: Abordagem empregada no MobiLine (MARINHO et al., 2012)

O MobiLine foi um projeto, de mesmo nome, desenvolvido através de uma parceria entre o Grupo de Redes, Engenharia de Software e Sistemas do Departamento de Computação da UFC e o Grupo de Reuso de Software COPPE da UFRJ. O objetivo desse projeto foi investigar as características existentes no desenvolvimento de software móvel e sensível ao contexto com a intenção de criar uma linha de produtos para esse domínio (MOBILINE, 2012).

Neste trabalho, será usado a LPSSC do nível específico (*"Guia de visita móveis"*. O produto configurado que é usado neste trabalho é o *"littleGreatTour"*.

6.1.1 Diagrama de característica

O diagrama de características da MobiLine é composto de:

- 12 características mandatórias (sendo uma delas a característica raiz da LPSSC),
- 23 características opcionais,
- 7 pontos de variação inclusivos (OU), e
- 1 ponto de variação exclusivo (Ou Exclusivo).

As características mandatórias são: *"Show Location", "Show Map", "Context Management", "Capture", "Acquisition", "Inference", "Message Exchange", "Service Description", "Show Environment Profile", "List Items" e "Show Item Profile"*. A característica raiz é a característica *"Mobile Guide"*.

As características opcionais são *"Show Event", "Set Profile", "Set Roadmap", "Persistency", "Synchronous", "RPC", "Tuple", "Event", "Show Profile Compatibility", "Security", "Authorization", "Authentication", "Centralized", "Distributed", "Hybrid", "State Based", "Logic Based", "Algebra Based", "KeyWord", "procedure Signature", "Video", "Image" e "Text"*.

Os pontos de variação inclusivos são *"Exchange Type", "Asynchronous", "Privacy", " Service Discovery", "Semantics", "Syntatic" e "Show Documents"*. O ponto de variação exclusivo é *"Application Layer"*.

Por conta da magnitude do diagrama de características do MobiLine, o diagrama de características é exibido por partes na Seção 6.1.5 (para ver o diagrama em uma só imagem consulte *"http://www.great.ufc.br/~pauloalexandre/FixTure/mobi_geral.png")*.

6.1.2 Regras de composição

A MobiLine possui 6 regras de composição, são elas:

- Regra de Composicao RC_9: (PRESENTE Optional: Centralized) → ((((PRESENTE Optional: KeyWord) AND (PRESENTE Optional: procedure Signature)) AND ((PRESENTE Optional: State Based) AND (PRESENTE Optional: Logic Based))) AND (PRESENTE Optional: Algebra Based))
- Regra de Composicao RC_{10}: (PRESENTE Optional: Distributed) → ((PRESENTE Optional: Algebra Based) AND (((PRESENTE Optional: Logic Based) AND (PRESENTE Optional: State Based)) AND ((PRESENTE Optional: KeyWord) AND (PRESENTE Optional: procedure Signature))))
- Regra de Composicao RC_{11}: (PRESENTE Optional: Hybrid) → ((PRESENTE Optional: Algebra Based) AND (((PRESENTE Optional: Logic Based) AND (PRESENTE Optional: State Based)) AND ((PRESENTE Optional: KeyWord) AND (PRESENTE Optional: procedure Signature))))
- Regra de Composicao RC_{12}: (PRESENTE Optional: Centralized) → ((PRESENTE Optional: RPC) AND (((PRESENTE Optional: Tuple) AND (PRESENTE Optional: RPC)) AND ((PRESENTE Optional: Event) AND (PRESENTE Optional: RPC))))
- Regra de Composicao RC_{13}: (PRESENTE Optional: Distributed) → ((PRESENTE Optional: RPC) AND (((PRESENTE Optional: RPC) AND (PRESENTE Optional: Tuple)) AND ((PRESENTE Optional: RPC) AND (PRESENTE Optional: Event))))
- Regra de Composicao RC_{14}: (PRESENTE Optional: Hybrid) → ((PRESENTE Optional: RPC) AND (((PRESENTE Optional: RPC) AND (PRESENTE Optional: Tuple)) AND ((PRESENTE Optional: RPC) AND (PRESENTE Optional: Event))))

- Regra de Composicao RC_{15}: PRESENTE Optional: Set Profile $\rightarrow$ PRESENTE Optional From Memory

6.1.3 Diagrama de contexto

A MobiLine possui as seguintes instâncias de *"EntidadedeContexto"*: *"Mobile Device", "User", "Item", "Environment" e "Network"*. Ademais, o diagrama de contexto possui as seguintes instâncias de *"InformacaoDeContexto"*: *"Mobile Device.Memory", "Mobile Device.Display", "Mobile Device.Libraries Available", "Mobile Device.Location", "Mobile Device.Battery", "User.Location", "User.Profile", "Item.Location", "Item.Profile", "Environment.Profile", "Environment.Location", "Mobile Device.CPU Usage", "Mobile Device . Mobility Level", "Network.Security", "Network.Type", "Network.Latency", "Network.Server"* e, por fim, *"Network.Ontology Representation"*.

Os relacionamentos entre estas instâncias podem ser vistos na Figura 6.2

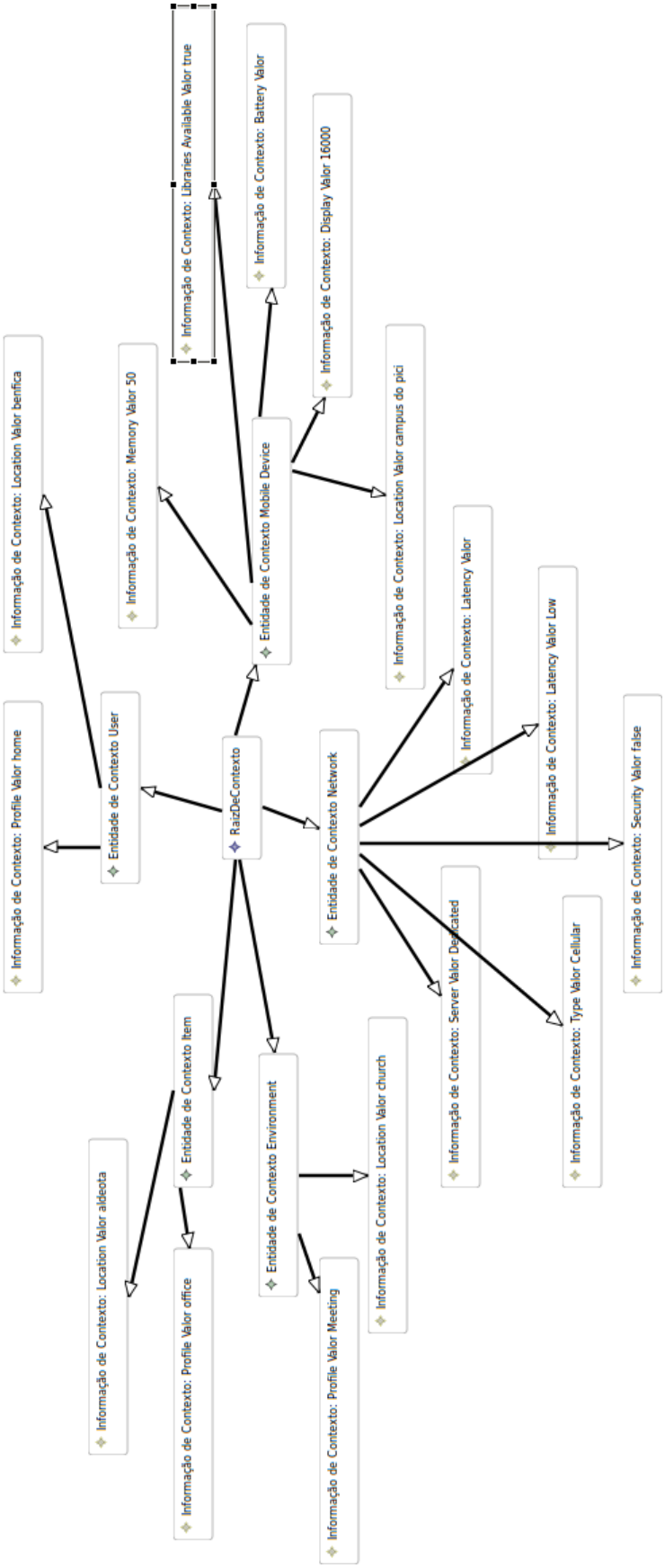


Figura 6.2: Diagrama de contexto do “Guia de Visita Móveis

6.1.4 Regras de contexto

A MobiLine possui as 13 regras de contexto citadas a seguir.

- Regra de contexto CTR_3: ((Network.Type IGUAL ADHOC) AND (Mobile Device.Mobility Level IGUAL HIGH)) → (PRESENTE Optional: Distributed)
- Regra de contexto CTR_5: ((Network.Type IGUAL ADHOC) AND (Mobile Device.Mobility Level IGUAL HIGH)) → (AUSENTE Optional: Centralized)
- Regra de contexto CTR_4: (Network.Type IGUAL ADHOC) → (PRESENTE Syntatic)
- Regra de contexto CTR_{12}: (Network.Type IGUAL Cellular) → (AUSENTE Optional: Distributed)
- Regra de contexto CTR_8: (Network.Type IGUAL ADHOC) → ((AUSENTE Optional: Event) AND (AUSENTE Optional: Tuple))
- Regra de contexto CTR_{13}: ((Network.Type IGUAL Cellular) AND ((Network.Server IGUAL Not Overloaded) AND (Network.Latency IGUAL LOW))) → (PRESENTE Optional: Centralized)
- Regra de contexto CTR_{14}: ((Network.Type IGUAL Cellular) AND ((Network.Server IGUAL Overloaded) AND (Network.Latency IGUAL High))) → (PRESENTE Optional: Hybrid)
- Regra de contexto CTR_{16}: ((Network.Type IGUAL Cellular) AND (Network.Server IGUAL Dedicated)) → (PRESENTE Semantics)
- Regra de contexto CTR_{17}: ((Network.Type IGUAL Cellular) AND ((Mobile Device.Battery MENORIGUAL 20) AND (Mobile Device.CPU Usage MAIORIGUAL 80))) → (PRESENTE Syntatic)
- Regra de contexto CTR_{18}: (Mobile Device.CPU Usage MAIORIGUAL 80) → (PRESENTE Syntatic)
- Regra de contexto CTR_{19}: (Network.Ontology Representation DIFERENTE false) → (PRESENTE Syntatic)
- Regra de contexto CTR_{22}: (Network.Type IGUAL Cellular) → (PRESENTE Optional: Centralized)
- Regra de contexto CTR_{24}: (Mobile Device.Libraries Available IGUAL true) → ((PRESENTE Optional: Text) AND (PRESENTE Optional: Image))

6.1.5 Relacionamentos

As características e pontos de variação expostos na Seção 6.1.1 apresentam os relacionamentos exibidos nas Figuras 6.3, 6.4 e 6.5.

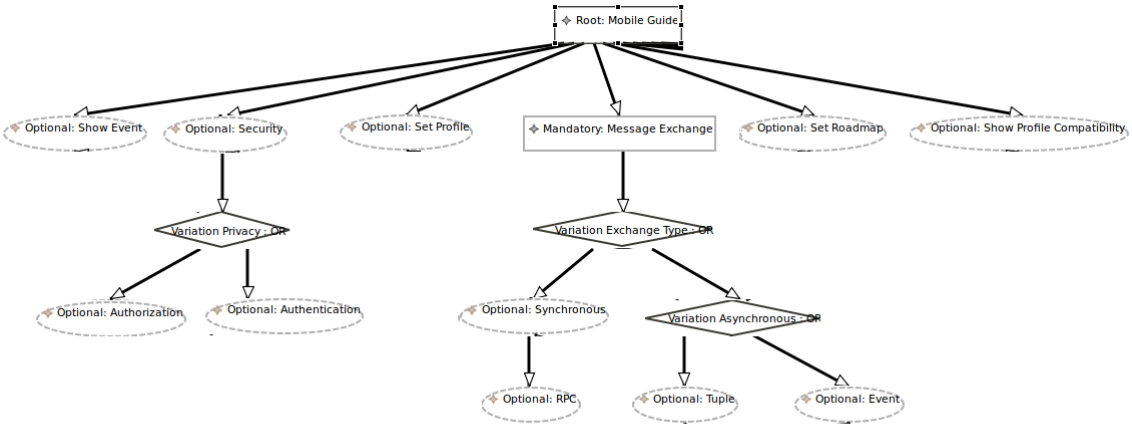


Figura 6.3: Diagrama de características do MobiLine (parte 1)

Na Figura 6.3, as características *"Show Event", "Security", "Set Profile", "Message Exchange", "Set Roadmap" e "Show Profile Compatibility"* são modeladas como características filhas da característica raiz *"Root"*.

A característica *"Security"* possui como característica filha um ponto de variação (OU) chamado *"Privacy"*, o qual, por vez, tem duas características opcionais associadas: *"Authorization"* e *"Authentication"*.

A característica *"Message Exchange"* tem como única característica filha um ponto de variação (OU) *"Exchange Type"*. O ponto de variação *"Exchange Type"* tem como características filhas a característica *"Synchronous"* e o ponto de variação *"Asynchronous"*. A característica *"Synchronous"* tem como característica filha a característica *"RPC"*. O ponto de variação *"Asynchronous"* tem como características filhas as características *"Tuple"* e *"Event"*.

Na Figura 6.4, é possível ver as características *"Show Location", "Context Management", "Show Map" e "Service Description"* como filhas da característica raiz *"Root"*. A característica *"Context Management"* tem como filhas as características *"Persistency"* e *"Capture"*. A característica *"Capture"* é pai da característica *"Acquisition"* que por sua vez é pai da característica *"Inference"*.

Ainda na Figura 6.4, a característica "Service Description" tem um ponto de variação (Ou Exclusivo) *"Application Layer"* como característica filha. As características *"Centralized"*, *"Distributed"* e *"Hybrid"* são as características filhas de *"Application Layer"*.

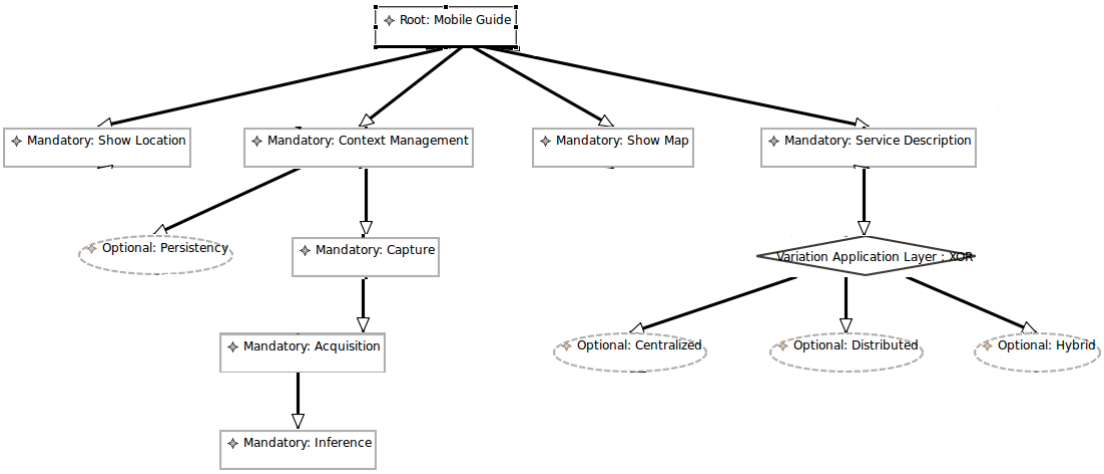


Figura 6.4: Diagrama de características do MobiLine (parte 2)

Na Figura 6.5, a característica *"Show Environment Profile"* é pai da característica *"List Items"*. A característica *"List Items"* é pai da característica *"Show Item Profile"*. Por sua vez, a característica *"Show Item Profile"* tem um ponto de variação (OU) associado chamado *"Show Documents"*; este ponto de variação tem como características filhas as características *"Video"*

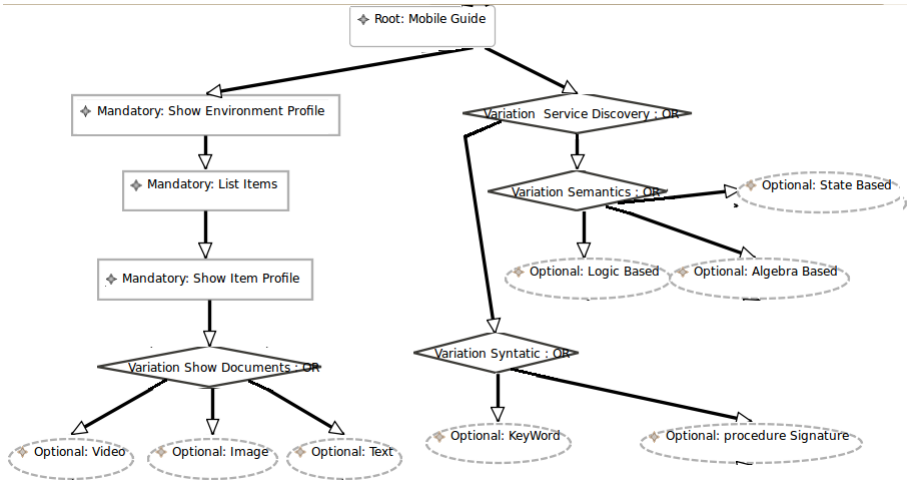


Figura 6.5: Diagrama de características do MobiLine (parte 3)

6.2 Metodologia

A avaliação preliminar apresentada neste capítulo seguiu a abordagem detalhada adiante. Inicialmente, o site do MobiLine (MOBILINE, 2012) foi consultado a fim de obter a documentação necessária. Conforme explicado na Seção 6.1, o MobiLine trabalha com dois níveis: o base e o específico. Como a avaliação preliminar irá trabalhar a nível específico (aplicações do tipo *"Guia de Visita Móveis"*), os documentos do nível específico do MobiLine foram coletadas e organizadas em diagramas de características, diagrama de contexto, regras de composição e regras de contexto.

Em seguida, cada um desses meta-modelos foram introduzidos na FixTure. Conforme relatado na Seção 6.3, não foi possível modelar na FixTure alguns dos elementos da modelagem original e essa impossibilidade se ocorreu pois a FixTure nem mesmo permitia a inclusão de certos erros de modelagens que estavam presentes na modelagem original como características com duas características pais.

Uma vez criados os arquivos representando os meta-modelos na FixTure, um produto foi configurado originando assim o *"littleGreatTour"* que é um produto genérico da LPSSC *"Guia De Vsita Móveis"*. Após a configuração de um produto, o PRECISE foi posto em execução usando o *"littleGreatTour"* como entrada para as fases 4 e 5. Para cada uma das fases do PRECISE, os erros que foram encontrados foram listados na Seção 6.3. A simulação foi destaca e uma seção é dedicada a ela.

Conforme exposto no Capítulo 5, este trabalho propôs dois comportamentos: em profundidade e em largura. Em ambos os comportamentos, a condição de parada envolvia a quantidade de transições totais (tanto as transições seguras como as inseguras). Essa condição de parada assumiu os seguintes valores 70, 80, 90, 100, 110, ... e 200 totalizando 14 execuções. Ou seja, quando a quantidade de transições chegava a esses valores, a execução doo algoritmo chegava ao fim.

Porém, é importante ressaltar que após a quarta execução (com 100 transições como condição de parada dos comportamentos) os grafos produzidos pelos dois comportamentos não sofreram alterações. Essa quantidade de transições máximas, a partir da qual não ocorrem mais alterações no grafo de simulação criado, é dependente da LPSSC e produto configurado considerados. Logo, este valor não pode ser assumido como universal.

6.3 Problemas identificados pelo PRECISE e Fixture

Durante a utilização da FixTure para especificar e validar a LPSSC *"Mobile Visit Guide"*, alguns elementos da modelagem presentes em (MOBILINE, 2012) não puderam ser transpostos para a notação da FixTure pois eram modelagens equivocadas. Por outro lado, outras modelagens também equivocadas puderam ser inseridas na FixTure porém esta foi capaz de detectá-las e evitar que erros consequentes da modelagem equivocada fossem propagados para as fases posteriores do PRECISE.

A característica *"Service Description"* modelada originalmente como característica mandatória e com cardinalidade 1-3 associada (indicando que pode conter no mínimo uma característica filha e no máximo três características filhas) foi modelada na FixTure como um ponto de variação do tipo *"OR"*. Tal modificação foi necessária, pois na modelagem proposta pela FixTure somente pontos de variação podem conter cardinalidades associadas. Pelo mesmo motivo, as seguintes alterações foram executadas:

- *"Application Layer 1-1"* de mandatória para ponto de variação XOR
- *"Semantics 1-1"* de opcional para ponto de variação XOR

- *"Syntatic 1-1"* de opcional para ponto de variação XOR
- *"Capture 1-3"* de opcional para ponto de variação OR
- *"Exchange Type 1-1"* de opcional para ponto de variação XOR
- *"Asynchronous 0-2"* de opcional para ponto de variação OR
- *"Privacy 0-1"* de opcional para ponto de variação XOR
- *"Show Documente 1-3"* de opcional para ponto de variação OR
- *"Show Documente 1-3"* de opcional para ponto de variação OR
- *"Text 1-2"* de mandatória para ponto de variação OR
- *"Image 0-2"* de mandatória para ponto de variação OR

Além disso, como as regras de boa formação definidas pelo PRECISE estipulam que os elementos de um diagrama de características tenham nomes distintos, a FixTure detectou que o ponto de variação *"Text"* e a característica opcional *"Text"* (que é filha do ponto de variação citado) possuem o mesmo nome. Uma vez apontado este problema, foi necessário alterar o nome do ponto de variação para *"PVText"*. De forma semelhante, o ponto de variação *"Asynchronous"* teve seu nome alterado para *"PVAsynchronous"* pelo fato de existir uma outra característica (a característica mandatória *"Asynchronous"*) com o mesmo nome.

A FixTure também detectou que outra regra de boa formação definida pelo PRECISE foi violada. Neste caso, trata-se das características, modeladas originalmente como *"mandatória"*, *"External Service"*, *"From Sensor"*, e *"From Memory"*. Elas são características mandatórias e, de acordo com as regras de boa formação do PRECISE, devem ser modeladas como filhas de características mandatórias ou da característica raiz. Essa regra de boa formação do PRECISE não é obedecida, pois as três são filhas de um ponto de variação (a saber, *"Capture"*). Para corrigir tal violação, as três características mandatórias foram remodeladas como características opcionais.

Ainda no que se refere ao diagrama de característica do *"Mobile Visit Guide"*, a FixTure detectou um ciclo presente entre as características mandatórias *"Show Map"* e *"Show Location"* e a característica raiz. *"Show Map"* e *"Show Location"* são filhas da característica raiz e *"Show Location"* estava modelada como filha também de *"Show Map"* configurando um ciclo. Como solução, *"Show Location"* está modelada sem *"Show Map"* como filha.

No tocante às regras de composição, apenas as duas regras de composição abaixo continham algum problema.

- Show Event $\rightarrow$ Access: *"Access"* é uma característica mandatória e de acordo com as regras de boa formação definidas no PRECISE, as características mandatórias não podem ser referenciadas em regras de composição.

- Show Documents → (Via Bluetooth OR External Service): *"Via Bluetooth"* não é uma característica do diagrama de características.

É importante salientar que, em ambos os casos, a ferramenta FixTure não permitiu a inserção dessas regras.

Em relação ao diagrama de contexto, os principais problemas foram a ausência de uma raiz de contexto que teria como filhas as entidades de contexto *"Mobile Device"*, *"User"*, *"Item"* e *"Environment"* e a ausência da indicação do tipo de dado das informações de contexto. Nesse caso, foram assumidos os seguintes tipos de dados para as informações de contexto:

- Memory: inteiro indicando a quantidade de memória
- Libraries Available: booleano indicando se há ou não bibliotecas disponíveis
- Display: inteiro indicando a quantidade de cores
- Battery: float indicando porcentagem da bateria
- Location: string indicando a posição
- Profile: string indicando o perfil carregado na aplicação
- Type: string indicando o tipo da rede
- Server: string que representa o status do servidor (*"overloaded"*, *"not overloaded"*, *"dedicated"*)
- Latency: string indicando a latência da rede (*"high"* e *"low"*)
- Security: booleano indicando se a rede é segura ou não

6.4 Simulação

Para realizar a simulação é criado um produto inicial chamado *"littleGreatTour"*, usando a representação simbólica apresentada no Capítulo 4. Como padrão para uma característica com nome *"x"*, vamos criar a variável V_x. Assim, são atribuídos os seguintes valores para as variáveis booleanas criadas para as características e atributos, conforme pode ser visto na Tabela 6.1:

A FixTure informa que há 71 formas de ativar as 12 regras de contexto de forma simultânea. Nesse capítulo, são exibidas 10 maneiras de ativar as regras de contexto. As formas de múltipla ativação pode ser consultadas integralmente no site do MobiLine (MOBILINE, 2012).

1. $CTR_{17}, CTR_{12}, CTR_{18}, CTR_{22}$
2. $CTR_{12}, CTR_{18}, CTR_{22}, CTR_{19}$

$V_{ShowLocation} = 1$	$V_{ShowMap} = 1$	$V_{ShowEvent} = 1$
$V_{ContextManagement} = 1$	$V_{Capture} = 1$	$V_{Acquisition} = 1$
$V_{Inference} = 1$	$V_{MessageExchange} = 1$	$V_{ExchangeType} = 1$
$V_{Asynchronous} = 1$	$V_{Tuple} = 1$	$V_{ShowProfileCompatibility} = 1$
$V_{Security} = 1$	$V_{Privacy} = 0$	$V_{Authentication} = 0$
$V_{ServiceDescription} = 1$	$V_{ServiceDiscovery} = 1$	$V_{Semantics} = 0$
$V_{AlgebraBased} = 0$	$V_{Syntatic} = 1$	$V_{procedureSignature} = 0$
$V_{ShowEnvironmentProfile} = 1$	$V_{ListItems} = 1$	$V_{ShowItemProfile} = 1$
$V_{ShowDocuments} = 1$	$V_{Image} = 1$	$V_{Text} = 1$
$V_{SetProfile} = 0$	$V_{SetRoadmap} = 0$	$V_{Persistency} = 0$
$V_{Synchronous} = 0$	$V_{RPC} = 0$	$V_{Event} = 0$
$V_{Authorization} = 0$	$V_{Centralized} = 0$	$V_{Distributed} = 0$
$V_{Hybrid} = 0$	$V_{StateBased} = 0$	$V_{LogicBased} = 0$
$V_{KeyWord} = 0$	$V_{Video} = 0$	

Tabela 6.1: Configuração do littleGreatTour

3. $CTR_{12}, CTR_{18}, CTR_{22}$
4. $CTR_{12}, CTR_{18}, CTR_{24}, CTR_{22}, CTR_{19}$
5. $CTR_{12}, CTR_{18}, CTR_{24}, CTR_{22}$
6. $CTR_{12}, CTR_{22}, CTR_{19}$
7. CTR_{12}, CTR_{22}
8. $CTR_{12}, CTR_{24}, CTR_{22}, CTR_{19}$
9. $CTR_{12}, CTR_{24}, CTR_{22}$
10. $CTR_{13}, CTR_{12}, CTR_{18}, CTR_{22}, CTR_{19}$

No comportamento em profundidade, as quantidades de transições foram 70, 80, 90 e 100 transições. Para o comportamento em largura, foram realizados quatro iterações. Em ambos os comportamentos, a saída da simulação (composta pelos estados e transições possíveis) pouco se alterou após a segunda execução. É importante observar que essa quantidade é específica à LPSSC modelada e ao produto escolhido como ponto de partida. Desta forma, esta quantidade (quatro) foi obtida empiricamente e que provavelmente é diferente para outro par LPSSC-produto inicial.

Para cada um dos comportamentos explicados no Capítulo 4, o PRECISE foi executado quatro vezes. Em cada uma dessas execuções incrementa-se a condição de parada: para o comportamento em profundidade, a condição de parada é a quantidade de transições enquanto que no comportamento em largura é quantidade de iterações. O número quatro foi obtido através de uma observação empírica. Na verdade, ambos os comportamento foram executados inúmeras vezes aumentando as quantidades de transições (70, 80, 90, 100, 110, ...). Percebeu-se que

após 100 transições, a simulação para o produto configuração não gerava um grafo diferente. É importante ressaltar, que este número quatro pode ser diferente se um produto configurado for dado como entrada para o processo de simulação.

São mostrados resultados de cada um dos comportamentos nas Figuras 6.7 e 6.6.

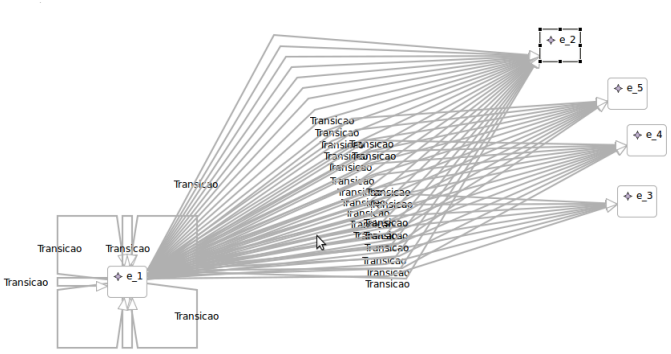


Figura 6.6: Simulação do produto *"littleGreatTour"* da *"MobiLine"* usando o comportamento 1

Conforme a Figura 6.6 mostra, para o comportamento 1, tem-se as seguintes transições: $[littleGreatTour \rightarrow p_1]$, $[littleGreatTour \rightarrow p_2]$, $[littleGreatTour \rightarrow p_3]$, $[littleGreatTour \rightarrow p_4]$ e $[littleGreatTour \rightarrow p_5]$.

É interessante observar que p_1, p_2, p_3, p_4 e p_5 são configurações que desobecem alguma regra de composição ou de boa formação. Por este motivo, *"littleGreatTour"* não conseguiu se adaptar. Uma hipótese, que é confirmada durante a simulação usando o comportamento em profundidade, é que a partir de *"littleGreatTour"* qualquer adaptação ou adapta para ele mesmo ou adapta para um produto desobediente.

As configurações de p_1, p_2, p_3, p_4, e p_5, são mostradas parcialmente, por questão de legibilidade. São mostradas apenas as diferenças em relação de *"littleGreatTour"* para as configurações citadas.

- p_1: $V_{tuple} = 0$
- p_2: $V_{ApplicationLayer} = 1, V_{Centralized} = 1$
- p_3: $V_{Semantics} = 1, V_{ApplicationLayer} = 1, V_{Centralized} = 1$
- p_4: $V_{Hybrid} = 1, V_{Semantics} = 1, V_{ApplicationLayer} = 1, V_{Centralized} = 1$
- p_5: $V_{tuple} = 0, V_{ApplicationLayer} = 1, V_{Distributed} = 1$

Para o comportamento em profundidade, é necessário considerar o seguinte fato. Como na MobiLine, há 71 combinações possíveis de regras de contexto, teríamos também 71 transições possíveis. A simulação se baseia em um AFN, logo, a partir de uma mesma configuração de produto pode haver mais de uma transição partindo dela. Constatou-se também que

de uma configuração para outra existem várias formas de realizar tal adaptação. Portanto, algumas combinações adaptam o produto *"littleGreatTour"* para o mesmo produto p_1, por exemplo. Neste sentido, é interessante ver a quantidade de transições que têm as mesmas configurações de produto inicial e final.

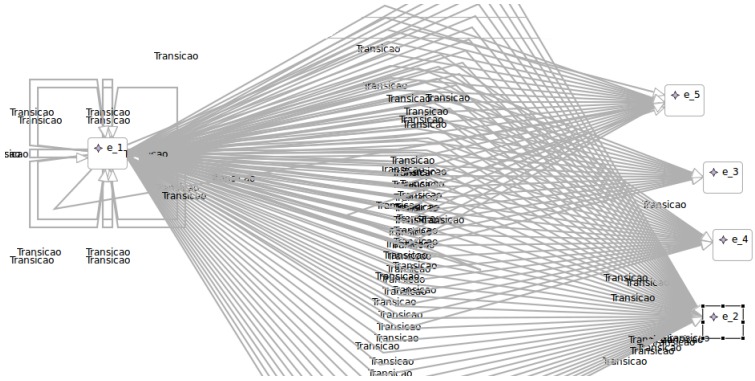


Figura 6.7: Simulação do produto *"littleGreatTour"* da *"MobiLine"* usando o comportamento 2

No comportamento em profundidade, temos as mesmas transições ocorridas nos primeiro passos da simulação com o comportamento em largura conforme a Figura 6.7 mostra. Como o comportamento 2 aplica todas as adaptações possíveis para um produto, será possível ver, para o produto inicial *"littleGreatTour"*, todas as configurações de produtos que podem ser alcançadas. Confirmando a hipótese levantada anteriormente, qualquer adaptação partindo de *"littleGreatTour"* ou não altera o seu estado (por exemplo, remover uma característica que o *"littleGreatTour"* não tem, não altera o produto *"littleGreatTour"*) ou leva para uma configuração de produto que é desobediente.

6.5 Conclusões

Este capítulo apresentou uma avaliação preliminar usando a ferramenta FixTure, onde a LPSSC escolhida foi o "Guia de Visita Móveis" oriundo da LPSSC MobiLine. Antes da aplicação do PRECISE em si, foram apresentados todos os elementos da MobiLine como as características mandatórias, opcionais e pontos de variação bem como os relacionamentos entre si. Além disso, as regras de composição foram expostas. O diagrama de contexto foi detalhado tal quais as regras de contexto.

Como o PRECISE necessita de um produto configurado para a sua execução, também foi definido o *"littleGreatTour"* para que o processo fosse executado completamente. A metodologia foi exposta neste capítulo ressaltando que, durante a inserção da LPSSC *"Guia de Visita Móveis"* na FixTure, algumas das modelagens equivocadas foram detectados antes da inserção das mesmas na FixTure enquanto que outras foram detectadas durante a execução do PRECISE em si. Uma listagem completa dos problemas foi oferecida neste capítulo.

Em seguida, o processo de simulação foi executado tomando o produto configurado *"littleGreatTour"* como ponto de partida da simulação em ambos os comportamentos (em lar-

gura e em profundidade) mostrando como resultados os grafos de simulação nos quais é possível ver as adaptações (tanto as seguras como as inseguras).

O próximo capítulo aborda as contribuições e as limitações deste trabalho como também aborda as conclusões às quais este trabalho chegou. Além disso, também sugere os prováveis futuros trabalhos a serem trilhados.

7 CONCLUSÃO

Neste capítulo, são apresentados as contribuições, resultados alcançados e as limitações deste trabalho de mestrado. Esta dissertação apresentou a implementação de um processo de verificação chamado PRECISE. Como forma de prover a automatização, a ferramenta FixTure foi construída. Esta ferramenta auxilia o engenheiro de software na criação e verificação dos modelos de características das LPSSCs bem como na configuração e adaptação de produtos desses modelos.

A Seção 7.1 lista as contribuições e resultados alcançados com essa dissertação de mestrado observando a premissa do presente trabalho e o estudo realizado no Capítulo 3 (Trabalhos relacionados). A Seção 7.2 complementa o presente capítulo ao discutir os trabalhos futuros que podem ser desenvolvidos tomando esse trabalho como ponto de partida.

7.1 Contribuições e resultados alcançados

Diagramas de características de uma LPSSC permitem ao engenheiro de software documentar os diversos produtos de uma LPSSC usando um só artefato, o próprio diagrama de características. Além disso, na necessidade de delinear as estruturas dos softwares estipulados no diagrama de características, o engenheiro pode especificar regras de composição que funcionam como restrições condicionais. Por exemplo, *"se o produto apresentar certa característica, então algumas outras características não devem estar presentes"*.

Essas regras de composição atuam sobre os produtos configurados de uma LPSSC os quais são sensíveis ao contexto. Portanto, mudanças nas informações de contexto podem acarretar em mudanças nos produtos. Assim, é necessário que haja uma modelagem do ambiente contextual e que sejam definidas as regras de adaptação (neste trabalho, chamadas de regras de contexto).

Enquanto na LPS tradicional a verificação de um único produto já era uma tarefa propensa a erros, verificar os diversos produtos que podem surgir das adaptações de uma LPSSC é uma tarefa mais complexa ainda. Nesse sentido, uma das principais contribuições deste trabalho foi a automatização do processo de verificação PRECISE em uma ferramenta de software, denominada FixTure. O processo de verificação PRECISE é o resultado de uma tese de doutorado (MARINHO, 2012) do Grupo de Redes, Engenharia de Software e Sistemas (GREat).

Para que fosse possível implementar o PRECISE, foi necessário que os meta-modelos apresentados no Capítulo 4 fossem desenvolvidos. Um dos resultados gerados pelo desenvolvimento desses meta-modelos mais concretos é a facilidade da construção de instâncias dos mesmos bem como a construção de relacionamentos de características mandatórias, opcionais e pontos de variação além de representar o ambiente contextual. Outro resultado gerado, é a possibilidade de implementar as verificações exigidas pelo PRECISE em EVL, que é uma linguagem de restrição próxima da OCL.

Contudo, uma limitação desses meta-modelos é que outras verificações (tais como

consulta de quantidade de produtos, checagem obediência de um produto em relação às regras de composição) requisitaram um poder computacional mais robusto. Nesse sentido, outra contribuição importante deste trabalho foi aprofundar a transformação dos diagramas de características e de contexto e das regras de composição e de contexto para lógica de primeira ordem. Na literatura, pode-se citar que (BENAVIDES; SEGURA; RUIZ-CORTÉS, 2010) e (TRINIDAD et al., 2008)) propuseram avanços significativos destas transformações, porém, nenhum dos dois trata elementos de contexto. Logo, uma lacuna foi preenchida pelo presente trabalho, pois trouxe a transformação do diagrama e das regras de contexto para fórmulas proposicionais. Adicionalmente, a transformação de atributos e regras de composição também teve, neste trabalho, grande atenção, pois, diferentemente de outros meta-modelos da literatura, os atributos são um dos grandes diferenciais do meta-modelo apresentado na tese de doutorado (MARINHO, 2012) a qual este presente trabalho dá suporte. Ainda no âmbito de oferecer um poder computacional mais robusto, pode-se citar a formalização das regras de contexto e composição usando a notação BNF.

Ainda neste âmbito, o presente trabalho considerou os campos de variação para cada atributo ou informação de contexto estipulado. A necessidade de considerar tais campos de variação é perceptível quando se verifica a satisfazibilidade das regras de composição em conjunção com os relacionamentos definidos pelo usuário no diagrama de características através da FixTure, pois é importante verificar se a LPSSC especificada pelo usuário contém pelo menos algum produto.

Outro ponto onde se percebe a importância de analisar os campos de variação é durante a simulação. Para a fase de simulação, foi necessário identificar quais regras de contexto podem ser ativadas simultaneamente em vez de considerá-las isoladamente. Considerar que as regras de contexto podem ser ativadas simultaneamente é importante por que torna a simulação mais próxima da realidade. Na verdade, para considerar a simultaneidade das regras de contexto também é necessário considerar o campo de variação das instâncias das regras de contexto para evitar que duas regras de contexto com eventos que não podiam ser ativadas simultaneamente fossem executadas ao mesmo tempo. Essa consideração pode ser entendida também como uma contribuição deste trabalho.

Com a simulação, também foi possível verificar um caminho de evolução de um produto da LPSSC sendo possível visualizar como um produto inicial se tornava instável após uma sequência de adaptações.

No âmbito das contribuições, é possível citar ainda o uso de um autômato finito não-determinístico (AFN) adaptado para LPSSC. Nesta direção, foi proposto um AFNL para representar o ciclo de vida de um produto de uma LPSSC.

Por fim, também podem ser citadas como resultados deste trabalho de pesquisa duas publicações: uma como principal autor (COSTA; ANDRADE, 2011) e outra (MARINHO et al., 2012) como terceiro autor. A primeira tratou explicitamente da implementação do mecanismo exposto neste trabalho e a segunda se concentrou em abordar a tese de doutorado (MARINHO, 2012) usando a FixTure como mecanismo de validação.

7.2 Trabalhos Futuros

A aplicação de um mecanismo de verificação diminui a quantidade de erros que podem surgir devido a modelagens equivocadas inseridas durante a especificação do diagrama de características ou do diagrama de contexto ou da regra de composição ou da regra de contexto.

Mesmo com a aplicação desse mecanismo, se essa tarefa não for automatizada os erros podem continuar existindo, pois a execução manual de um mecanismo é cansativa e propensa a erros.

Contudo, a automatização do mecanismo também não irá garantir a eliminação de todos os erros. Devido à grande quantidade de produtos esperados em uma LPSSC e de erros que surgem somente em configurações específicas de produtos, algumas lacunas permanecem abertas na análise automática de uma LPSSC. Neste sentido, são apresentadas algumas possibilidades de trabalhos futuros.

- paralelização: para os comportamentos descritos no final do Capítulo 4 (em especial o comportamento 2), é interessante paralelizar a simulação de forma a aumentar o desempenho da mesma e diminuir o tempo de simulação;
- escalabilidade: embora os comportamentos descritos no final do Capítulo 4 sejam decidíveis e o MobiLine tenha sido um estudo de caso que testou a aplicabilidade da FixTure, é interessante realizar testes em LPSSC de maiores tamanhos e analisar pontos de gargalo e, consequentemente, possíveis pontos de melhoria na ferramenta;
- continuação de simulação: em vez de produzir simulações isoladamente, dado o tamanho significativo de até $\mathcal{O}(2^n)$ possíveis produtos em uma LPSSC com n características opcionais seria interessante anexar o resultado de uma simulação com o de outra realizada em um momento diferente. O resultado direto dessa junção seria que teríamos um grafo desconexo como resultado da simulação. Aliada com a paralelização, a continuação de simulação permitiria aumentar a área de abrangência do grafo de simulação;
- novos algoritmos para comportamentos: este presente trabalho apresentou dois comportamentos para a simulação cujo resultado consistia de um grafo conexo para demonstrar como um produto partia de uma configuração para outra. Entretanto, diversos outros comportamentos podem ser propostos e por isso este trabalho é diretamente expansível por qualquer pessoa que queira propor um novo comportamento para a simulação;
- permitir que as regras de composição e de contexto possibilitem comparar atributos de características diferentes: na implementação atual da FixTure ao especificarmos uma expressão relacional (usando os operadores matemáticos relacionais $>, <, \leq, \geq, =$ e $\neq$) só é permitido compararmos o atributo ou a informação de contexto com algum valor especificado pelo usuário através da FixTure. Um grande incremento no poder de expressividade das LPSSCs especificadas na FixTure seria permitir comparações entre atributos ou entre informações de contexto. Por exemplo, suponha duas informação de contexto, uma chamada SinalChip1.Intensidade e outra SinalChip2.Intensidade; com essa possível adição

seria possível escrever regras de adaptação como *"se SinalChip1.Intensidade > SinalChip2.Intensidade então ..."*;

- melhorias na interface: a edição de regras de composição e contexto podem ser melhorados no sentido de propor que o usuário especifique as regras de composição e de contexto de forma textual em vez de forma gráfica, pois diminuiria o trabalho do usuário no sentido que não seria mais necessário especificar as regras de composição e de contexto arrastando item a item os elementos das regras;

- uso completo de lógica de primeira ordem: esta implementação do PRECISE especificou as regras de boa formação em EVL. Contudo, ao longo do desenvolvimento deste trabalho, viu-se a possibilidade de especificar as regras de boa formação também em lógica de primeira ordem. Por este ponto de vista, seria interessante analisar o desempenho da FixTure usando somente lógica de primeira ordem. Esta abordagem foi empregada na validação manual apresentada em (MARINHO, 2012);

- provisão de mensagens de erro e formas de correção: de acordo com (BENAVIDES; SEGURA; RUIZ-CORTÉS, 2010), uma das operações de análise automática de diagramas de características são as explanações. Neste sentido, seria interessante que a FixTure fosse enriquecida com mais mensagens explicativas, e em alguns casos, com ações de correção do erro. O framework em que a parte gráfica da FixTure foi desenvolvida ((EPSILON, 2012)) permite a realização de consertos dos problemas detectados. Logo, uma expansão da implementação atual seria a implementação da correção dos erros detectados; e

- criar uma versão web online da ferramenta: uma versão online ofereceria um ambiente mais expansível ainda e com a possibilidade de usar controles gráficos mais ricos. Além disso, por ser online, poderia ser criado uma repositório onde usuários poderiam compartilhar as LPSSCs de forma semelhante ao que a ferramenta (S.P.L.O.T, 2012) oferece.

Assim, a automatização apresentada nessa dissertação é um primeiro passo em direção a automatização completa do processo de verificação do modelos de características de uma LPSSC. Os trabalhos futuros apresentados acima são candidatos a estudos ou dissertações de mestrado. Dado que o número de operações de verificação cresce à medida que novos estudos são publicados, a área de automatização para a verificação destes modelos é próspera (BENAVIDES; SEGURA; RUIZ-CORTÉS, 2010).

REFERÊNCIAS BIBLIOGRÁFICAS

ABOWD, G.; DEY, A.; BROWN, P.; DAVIES, N.; SMITH, M.; STEGGLES, P. Towards a better understanding of context and context-awareness. In: GELLERSEN, H.-W. (Ed.). *Handheld and Ubiquitous Computing*. [S.l.]: Springer Berlin / Heidelberg, 1999, (Lecture Notes in Computer Science, v. 1707). p. 304–307. ISBN 978-3-540-66550-2.

ALMEIDA, E. de; ALVARO, A.; LUCREDIO, D.; GARCIA, V.; MEIRA, S. de L. A survey on software reuse processes. In: *Information Reuse and Integration, Conf, 2005. IRI -2005 IEEE International Conference on.* [S.l.: s.n.], 2005. p. 66 – 71.

ANTKIEWICZ, M.; CZARNECKI, K. Featureplugin: feature modeling plug-in for eclipse. In: *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. New York, NY, USA: ACM, 2004. (eclipse '04), p. 67–72.

BACHMANN, F.; BASS, L. Managing variability in software architectures. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 26, p. 126–132, May 2001. ISSN 0163-5948.

BAILIN, S. Kaptur: a tool for the preservation and use of engineering legacy. *CTA Incorporated*, 1992.

BENAVIDES, D.; SEGURA, S.; RUIZ-CORTÉS, A. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, Elsevier Science Ltd., Oxford, UK, UK, v. 35, n. 6, p. 615–636, Setembro 2010. ISSN 0306-4379.

BEUCHE, D.; PAPAJEWSKI, H.; SCHRöDER-PREIKSCHAT, W. Variability management with feature models. *Science of Computer Programming*, v. 53, n. 3, p. 333 – 352, 2004. ISSN 0167-6423.

BPMN. *BPMN - Business Process Model And Notation 2.0*. 2012. Disponível em: http://www.omg.org/spec/BPMN/2.0/. Acessado em 17/07/2012.

BUDINSKY, F. *Eclipse modeling framework: a developer's guide*. [S.l.]: Addison-Wesley Professional, 2004.

CETINA, C.; FONS, J.; PELECHANO, V. Applying software product lines to build autonomic pervasive systems. In: *Software Product Line Conference, 2008. SPLC '08. 12th International*. [S.l.: s.n.], 2008. p. 117–126.

CLEMENTS, P. Being proactive pays off. *Software, IEEE*, v. 19, n. 4, p. 28–30, jul/aug 2002. ISSN 0740-7459.

CLEMENTS, P.; NORTHROP, L. *Software product lines*. [S.l.]: Addison-Wesley, 2001.

CLEMENTS, P.; NORTHROP, L. *Software product lines: practices and patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0-201-70332-7.

COSTA, P. A. da S.; ANDRADE, R. M. de C. Uma ferramenta para verificação de consistência e integridade do modelo de características de linhas de produtos de software sensíveis ao contexto baseado em perfis da uml. In: *I Workshop de Teses e Dissertações do CBSoft*. [S.l.: s.n.], 2011. v. 6. ISSN 2178-6097.

CZARNECKI, K.; ANTKIEWICZ, M. Mapping features to models: A template approach based on superimposed variants. In: GLÜCK, R.; LOWRY, M. (Ed.). *Generative Programming and Component Engineering*. [S.l.]: Springer Berlin / Heidelberg, 2005, (Lecture Notes in Computer Science, v. 3676). p. 422–437. ISBN 978-3-540-29138-1.

CZARNECKI, K.; EISENECKER, U. W. *Generative programming: methods, tools, and applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 0-201-30977-7.

DEY, A. K. Understanding and using context. *Personal Ubiquitous Comput.*, Springer-Verlag, London, UK, v. 5, p. 4–7, January 2001. ISSN 1617-4909.

DEY, A. K.; ABOWD, G.; PINKERTON, M.; WOOD, A. Cyberdesk: a framework for providing self-integrating ubiquitous software services. In: *Proceedings of the 10th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM, 1997. (UIST '97), p. 75–76. ISBN 0-89791-881-9.

ECLIPSE. *The Eclipse Foundation open source community website*. 2012. Disponível em: http://eclipse.org/. Acessado em: 26/07/2012.

EMF. *Eclipse Modelling - EMF- Home*. 2011. Disponível em: http://www.eclipse.org/modeling/emf/. Acessado em: 20/07/2011.

EMFATIC. *Emfatic Language Reference*. 2012. Disponível em: http://www.eclipse.org/epsilon/doc/articles/emfatic/. Acessado em: 17/10/2012.

EPSILON. *Epsilon - Home Page*. 2012. Disponível em: http://www.eclipse.org/epsilon/. Acessado em: 22/07/2012.

EVL. *Eclipse Validation Language*. 2012. Disponível em: http://www.eclipse.org/epsilon/doc/evl/. Acessado em: 19/12/2012.

FAYAD, M. E.; SCHMIDT, D. C. Lessons learned building reusable oo frameworks for distributed software. *Commun. ACM*, ACM, New York, NY, USA, v. 40, n. 10, p. 85–87, out. 1997. ISSN 0001-0782.

FEATURE, C. *Captain Feature - Home Page*. 2012. Disponível em: http://sourceforge.net/projects/captainfeature/. Acessado em: 26/07/2012.

FEATUREIDE. *FeatureIDE*. 2011. Disponível em: http://wwwiti.cs.uni-magdeburg.de/iti_-db/research/featureide/. Acessado em: 26/07/2011.

FEATUREMAPPER. *FeatureMapper*. 2011. Disponível em: http://featuremapper.org/. Acessado em: 29/07/2011.

FERNANDES, P.; WERNER, C. M. L. Ubifex: Modeling context-aware software product lines. In: THIEL, S.; POHL, K. (Ed.). *SPLC (2)*. [S.l.]: Lero Int. Science Centre, University of Limerick, Ireland, 2008. p. 3–8.

FRAKES, W.; ISODA, S. Success factors of systematic reuse. *Software, IEEE*, v. 11, n. 5, p. 14–19, sep 1994. ISSN 0740-7459.

FRAKES, W.; KANG, K. Software reuse research: status and future. *Software Engineering, IEEE Transactions on*, v. 31, n. 7, p. 529–536, july 2005. ISSN 0098-5589.

FRAKES, W.; PRIETO-;DIAZ, R.; FOX, C. Dare: Domain analysis and reuse environment. *Annals of Software Engineering*, Springer Netherlands, v. 5, p. 125–141, 1998. ISSN 1022-7091.

FRAKES, W. B.; FOX, C. J. Sixteen questions about software reuse. *Commun. ACM*, ACM, New York, NY, USA, v. 38, n. 6, p. 75–ff., jun. 1995. ISSN 0001-0782.

GRISS, M. L. Software reuse: From library to factory. *IBM Systems Journal*, v. 32, n. 4, p. 548–566, 1993. ISSN 0018-8670.

HABERMANN, A. N.; FLON, L.; COOPRIDER, L. Modularization and hierarchy in a family of operating systems. *Commun. ACM*, ACM, New York, NY, USA, v. 19, n. 5, p. 266–272, maio 1976. ISSN 0001-0782.

HALLSTEINSEN, S.; HINCHEY, M.; PARK, S.; SCHMID, K. Dynamic software product lines. *Computer*, v. 41, n. 4, p. 93–95, april 2008. ISSN 0018-9162.

HALLSTEINSEN, S.; STAV, E.; SOLBERG, A.; FLOCH, J. Using product line techniques to build adaptive systems. In: *Software Product Line Conference, 2006 10th International*. [S.l.: s.n.], 2006. p. 150–159.

HEIDENREICH, F. Towards systematic ensuring well-formedness of software product lines. In: *Proceedings of the First International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2009. (FOSD '09), p. 69–74. ISBN 978-1-60558-567-3.

HEIDENREICH, F.; KOPCSEK, J.; WENDE, C. Featuremapper: mapping features to models. In: *Companion of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008. (ICSE Companion '08), p. 943–944. ISBN 978-1-60558-079-1.

JDD. *The JDD Project*. 2012. Disponível em: http://javaddlib.sourceforge.net/jdd/. Acessado em: 09/09/2012.

KANG, K.; COHEN, S.; HESS, J.; NOVAK, W.; PETERSON, A. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. [S.l.], 1998.

KEPHART, J.; CHESS, D. The vision of autonomic computing. *Computer*, v. 36, n. 1, p. 41 – 50, jan 2003. ISSN 0018-9162.

KRUEGER, C. Eliminating the adoption barrier. *Software, IEEE*, v. 19, n. 4, p. 29 –31, july-aug. 2002. ISSN 0740-7459.

KRUEGER, C. W. Software reuse. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 24, n. 2, p. 131–183, jun. 1992. ISSN 0360-0300.

KRUEGER, C. W. Biglever software gears and the 3-tiered spl methodology. In: *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. New York, NY, USA: ACM, 2007. (OOPSLA '07), p. 844–845. ISBN 978-1-59593-865-7.

LAGUNA, M.; GONZALEZ-BAIXAULI, B.; LOPEZ, O.; GARCIA, F. Introducing systematic reuse in mainstream software process. In: *Euromicro Conference, 2003. Proceedings. 29th*. [S.l.: s.n.], 2003. p. 351–358. ISSN 1089-6503.

LEE, J.; KANG, K. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In: *Software Product Line Conference, 2006 10th International*. [S.l.: s.n.], 2006. p. 10 pp. –140.

LISBOA, L.; GARCIA, V.; ALMEIDA, E.; MEIRA, S. Toolday: a tool for domain analysis. *International Journal on Software Tools for Technology Transfer*, Springer-Verlag, v. 13, n. 4, p. 337–353, 2011. ISSN 1433-2779.

LOUDEN, K. C. *Compiladores: princípios e práticas*. 1. ed. São Paulo: Pioneira Thomson Learnin, 2004. ISBN 85-221-0422-0.

MARINHO, F.; ANDRADE, R.; WERNER, C. A verification mechanism of feature models for mobile and context-aware software product lines. In: *Software Components, Architectures and Reuse (SBCARS), 2011 Fifth Brazilian Symposium on*. [S.l.: s.n.], 2011. p. 1 –10.

MARINHO, F.; COSTA, A.; LIMA, F.; NETO, J.; FILHO, J.; ROCHA, L.; DANTAS, V.; ANDRADE, R.; TEIXEIRA, E.; WERNER, C. An architecture proposal for nested software product lines in the domain of mobile and context-aware applications. In: *Software Components, Architectures and Reuse (SBCARS), 2010 Fourth Brazilian Symposium on*. [S.l.: s.n.], 2010. p. 51–60.

MARINHO, F. G. A proposal for consistency checking in dynamic software product line models using ocl. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. New York, NY, USA: ACM, 2010. (ICSE '10), p. 333–334. ISBN 978-1-60558-719-6.

MARINHO, F. G. *PRECISE - Processo de veRificação Formal para modElos de CaracterístIcas de Aplicações Móveis e Sensíveis ao ContExto*. Tese (Doutorado) — Universidade Federal Do Ceará, Brasil, Ceará, Agosto 2012.

MARINHO, F. G.; ANDRADE, R. M.; WERNER, C.; VIANA, W.; MAIA, M. E.; ROCHA, L. S.; TEIXEIRA, E.; FILHO, J. B. F.; DANTAS, V. L.; LIMA, F.; AGUIAR, S. Mobiline: A nested software product line for the domain of mobile and context-aware applications. *Science of Computer Programming*, n. 0, p. –, 2012. ISSN 0167-6423.

MARINHO, F. G.; LIMA, F.; FILHO, J. a. B. F.; ROCHA, L.; MAIA, M. E. F.; AGUIAR, S. B. de; DANTAS, V. L. L.; VIANA, W.; ANDRADE, R. M. C.; TEIXEIRA, E.; WERNER, C. A software product line for the mobile and context-aware applications domain. In: *Proceedings of the 14th international conference on Software product lines: going beyond*. Berlin, Heidelberg: Springer-Verlag, 2010, (SPLC'10). p. 346–360. ISBN 3-642-15578-2, 978-3-642-15578-9.

MARINHO, F. G.; MAIA, P. H. M.; ANDRADE, R. M. C.; VIDAL, V. M. P.; COSTA, P. A. S.; WERNER, C. Safe adaptation in context-aware feature models. In: *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2012. (FOSD '12), p. 54–61. ISBN 978-1-4503-1309-4.

MASSEN, T. von der; LICHTER, H. Requiline: A requirements engineering tool for software product lines. In: LINDEN, F. van der (Ed.). *Software Product-Family Engineering*. [S.l.]: Springer Berlin / Heidelberg, 2004, (Lecture Notes in Computer Science, v. 3014). p. 168–180. ISBN 978-3-540-21941-5.

MCGREGOR, J.; NORTHROP, L.; JARRAD, S.; POHL, K. Initiating software product lines. *Software, IEEE*, v. 19, n. 4, p. 24–27, jul/aug 2002. ISSN 0740-7459.

MCILROY, M.; BUXTON, J.; NAUR, P.; RANDELL, B. Mass produced software components. *Software Engineering Concepts and Techniques*, NATO Science Committee, p. 88–98, 1969.

MCKINLEY, P.; SADJADI, S.; KASTEN, E.; CHENG, B. Composing adaptive software. *Computer*, v. 37, n. 7, p. 56 – 64, july 2004. ISSN 0018-9162.

MDT. *Eclipse Modelling - MDT- Home*. 2011. Disponível em: http://www.eclipse.org/modeling/mdt/. Acessado em: 20/07/2011.

MEYER, M.; LEHNERD, A. *The Power of Product Platforms*. [S.l.]: Free Press, 1997.

MOBILINE. *MobiLine - A Software Product Line for the Development of Mobile and Context-Aware Applications*. 2012. Disponível em: http://mobiline.great.ufc.br/index.php. Acessado em 20/06/2012.

MORISIO, M.; EZRAN, M.; TULLY, C. Success and failure factors in software reuse. *Software Engineering, IEEE Transactions on*, v. 28, n. 4, p. 340 –357, apr 2002. ISSN 0098-5589.

NEIGHBORS, J. M. The draco approach to constructing software from reusable components. *Software Engineering, IEEE Transactions on*, SE-10, n. 5, p. 564 –574, sept. 1984.

NORTHROP, L. M. SEI's software product line tenets. *IEEE Software*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 19, p. 32–40, July 2002. ISSN 0740-7459.

OREIZY, P.; GORLICK, M.; TAYLOR, R.; HEIMHIGNER, D.; JOHNSON, G.; MEDVIDOVIC, N.; QUILICI, A.; ROSENBLUM, D.; WOLF, A. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications, IEEE*, v. 14, n. 3, p. 54–62, may/jun 1999. ISSN 1094-7167.

OREIZY, P.; MEDVIDOVIC, N.; TAYLOR, R. N. Runtime software adaptation: framework, approaches, and styles. In: *Companion of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008. (ICSE Companion '08), p. 899–910. ISBN 978-1-60558-079-1.

PAPYRUS. *Papyrus*. 2011. Disponível em: http://www.eclipse.org/modeling/mdt/papyrus/. Acessado em: 26/07/2011.

PARNAS, D. On the design and development of program families. *Software Engineering, IEEE Transactions on*, SE-2, n. 1, p. 1–9, march 1976. ISSN 0098-5589.

PARRA, C.; BLANC, X.; DUCHIEN, L. Context awareness for dynamic service-oriented product lines. In: *Proceedings of the 13th International Software Product Line Conference*. Pittsburgh, PA, USA: Carnegie Mellon University, 2009. (SPLC '09), p. 131–140.

POHL, K.; BöCKLE, G.; LINDEN, F. J. v. d. *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN 3540243720.

POSLAD, S. *Front Matter*. [S.l.]: John Wiley & Sons, Ltd, 2009. ISBN 9780470779446.

PRIETO-DIAZ, R. Status report: software reusability. *Software, IEEE*, v. 10, n. 3, p. 61–66, may 1993. ISSN 0740-7459.

SAMA, M.; ELBAUM, S.; RAIMONDI, F.; ROSENBLUM, D.; WANG, Z. Context-aware adaptive applications: Fault patterns and their automated identification. *Software Engineering, IEEE Transactions on*, v. 36, n. 5, p. 644 –661, sept.-oct. 2010. ISSN 0098-5589.

SCHILIT, B.; ADAMS, N.; WANT, R. Context-aware computing applications. In: *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*. [S.l.: s.n.], 1994. p. 85–90.

SCHILIT, B.; THEIMER, M. Disseminating active map information to mobile hosts. *Network, IEEE*, v. 8, n. 5, p. 22–32, sep/oct 1994. ISSN 0890-8044.

SCHMIDT, D.; BUSCHMANN, F. Patterns, frameworks, and middleware: their synergistic relationships. In: *Software Engineering, 2003. Proceedings. 25th International Conference on*. [S.l.: s.n.], 2003. p. 694–704. ISSN 0270-5257.

SPINCZYK, O.; BEUCHE, D. Modeling and building software product lines with eclipse. In: *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2004. (OOPSLA '04), p. 18–19. ISBN 1-58113-833-4.

S.P.L.O.T. *Software Product Line Online Tools*. 2012. Disponível em: http://gsd.uwaterloo.ca:8088/SPLOT/index.html. Acessado em: 01/09/2012.

THAO, C.; MUNSON, E. V.; NGUYEN, T. N. Software configuration management for product derivation in software product families. In: . Los Alamitos, CA, USA: IEEE Computer Society, 2008. v. 0, p. 265–274. ISBN 978-0-7695-3141-0.

TRACZ, W. Why reusable software isn't. In: *North Carolina Univ, Proceedings of the Workshop on Future Directions in Computer Architecture and Software p 171-177(SEE N 88-18191 10-60)*. [S.l.: s.n.], 1986.

TRINIDAD, P.; BENAVIDES, D.; DURáN, A.; RUIZ-CORTéS, A.; TORO, M. Automated error analysis for the agilization of feature modeling. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 81, p. 883–896, Junho 2008. ISSN 0164-1212.

UML. *Object Management Group - UML*. 2011. Disponível em: http://www.uml.org/. Acessado em: 20/07/2011.

WANT, R.; HOPPER, A.; aO, V. F.; GIBBONS, J. The active badge location system. *ACM Trans. Inf. Syst.*, ACM, New York, NY, USA, v. 10, n. 1, p. 91–102, jan. 1992. ISSN 1046-8188.

WEISER, M. The computer for the 21st century. *Scientific American*, v. 265, n. 3, p. 66–75, January 1991.

WEISER, M. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 3, p. 3–11, July 1999. ISSN 1559-1662.

XFEATURE. *XFeature – Home*. 2011. Disponível em: http://www.pnp-software.com//XFeature/. Acessado em: 26/07/2011.