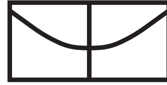

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Flexibilidade em Linhas de Produtos Dinâmicas Cientes de Qualidade: uma Abordagem Baseada em Linguagens Específicas de Domínio

Leonardo Monteiro Pessoa

Brasília
2014

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

Flexibilidade em Linhas de Produtos Dinâmicas Cientes de Qualidade: uma Abordagem Baseada em Linguagens Específicas de Domínio

Leonardo Monteiro Pessoa

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientador
Prof. Dr. Vander Alves

Brasília
2014

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

Coordenadora: Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo

Banca examinadora composta por:

Prof. Dr. Vander Alves (Orientador) — CIC/UnB
Prof. Dr. Fernando José Castor de Lima Filho — CIn/UFPE
Prof.ª Dr.ª Genaína Nunes Rodrigues — CIC/UnB

CIP — Catalogação Internacional na Publicação

Pessoa, Leonardo Monteiro.

Flexibilidade em Linhas de Produtos Dinâmicas Cientes de Qualidade: uma Abordagem Baseada em Linguagens Específicas de Domínio / Leonardo Monteiro Pessoa. Brasília : UnB, 2014.
87 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2014.

1. Linha de produto de software dinâmica, 2. Linguagem específica de domínio, 3. Qualidade de serviço

CDU 004.4

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

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

Flexibilidade em Linhas de Produtos Dinâmicas Cientes de Qualidade: uma Abordagem Baseada em Linguagens Específicas de Domínio

Leonardo Monteiro Pessoa

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

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

Prof. Dr. Fernando José Castor de Lima Filho
CIn/UFPE

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

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo
Coordenadora do Mestrado em Informática

Brasília, 7 de julho de 2014

Dedicatória

Aos felinos.

Agradecimentos

Inicialmente, gostaria de expor minha gratidão a todas as pessoas que, em maior ou menor grau, contribuíram para a minha formação e me estimularam a prosseguir nos estudos. Em especial, gostaria de agradecer meus pais pelo amor, carinho, compreensão e incentivo em todas as etapas da minha vida.

Gostari de agradecer ao meu orientador Vander por acreditar no meu potencial e pelo tempo e atenção dispensados no desenvolvimento deste trabalho.

Agradeço ao prof. Czarnecki pelas sugestões que contribuíram para este trabalho, e ao Dr. Hervaldo pelas contribuições para este trabalho como especialista no domínio.

Agradeço também a Graziella, pois sem seus batimentos cardíacos este trabalho não seria possível.

Agradeço à minha noiva, Carla, pelo todo seu carinho, compreensão e cuidados nesses últimos meses. Você foi minha força para continuar em meio às dificuldades.

Muito obrigado a todos.

Resumo

De modo a acomodar necessidades específicas, certos sistemas precisam ser reconfigurados durante sua execução, através da substituição de componentes responsáveis por essas necessidades de acordo com o contexto atual da aplicação. Tais sistemas são classificados como Linhas de Produto de Software Dinâmicas (LPSD). Entretanto, em alguns domínios, as regras para tal reconfiguração podem elas próprias ser variáveis de tal forma que é impossível de ser comportada por uma simples parametrização numérica tanto no tempo quanto no espaço. Dado este contexto, o presente trabalho apresenta uma proposta para resolução do problema de reconfiguração dinâmica de LPSDs através do uso de uma linguagem específica de domínio. Para validação deste trabalho, esta solução será implementada numa LPSD de uma Rede de Sensores do Corpo Humano (RSCH) sobre a qual será realizada uma comparação da análise de desempenho e de *safety*.

Palavras-chave: Linha de produto de software dinâmica, Linguagem específica de domínio, Qualidade de serviço

Abstract

In order to accommodate specific needs, certain systems have to be reconfigured during runtime through the replacement of components responsible for such needs according to the current context of the application. Such systems are classified as Dynamic Software Product Lines (DSPL). However, in certain domains, the rules used for this reconfiguration can themselves be variable such that is impossible to contain with a simple numeric parameterisation in both time and space. Given this context, the present work presents a proposal to resolve the problem of dynamic reconfiguration of DSPLs through the use of a domain-specific language (DSL). To validate this work, this solution will be implemented on a DSPL of a Body Sensor Network (BSN) over which will be made performance and safety analyses.

Keywords: Dynamic software product lines, Domain-specific language, Quality of service

Sumário

Lista de Figuras **x**

Lista de Tabelas **xii**

Lista de Listagens **xiii**

1 Introdução **1**

1.1 Definição do Problema 1
1.2 Solução Proposta 2
1.3 Avaliação 3
1.4 Resumo de Contribuições 3
1.5 Organização do Trabalho 4

2 Fundamentação Teórica **5**

2.1 Linha de Produtos de Software 5
2.1.1 Tempo de Ligação 6
2.1.2 Linha de Produtos de Software Dinâmica 7
2.2 Rede de Sensores do Corpo Humano 9
2.2.1 O Aplicativo Monitor 10
2.3 Linguagens Específicas de Domínio 12
2.3.1 Processo de Definição 14
2.4 Máquinas Virtuais 15
2.4.1 Implementação da VM 16
2.4.2 Pequenas Plataformas 17
2.5 Resumo 17

3 Modelagem do Problema **18**

3.1 Motivação 18
3.2 Cenários 19
3.3 Requisitos 20

3.4 Resumo 21

4 Evolução de Linhas de Produtos de Software Dinâmicas **22**

4.1 O Processo de Desenvolvimento 23

4.1.1 Modelagem de Contexto e Objetivos 25

4.2 Linguagem Específica do Domínio 27

4.2.1 Definição da Sintaxe Abstrata 29

4.2.2 Definição da Sintaxe Concreta 31

4.2.3 Definindo o Comportamento da DSL 36

4.3 Análise de Objetivos de Qualidade 37

4.4 Arquitetura 38

4.4.1 Definição da Máquina Virtual 40

4.4.2 O Processo de Compilação da DSL 43

4.5 Implementação 45

4.5.1 Aplicativo Móvel 46

4.5.2 Servidor de Aplicação WEB 47

4.6 Resumo 51

5 Avaliação **52**

5.1 *Goal Question Metric* (GQM) 52

5.1.1 Questões e Métricas 53

5.2 Contexto e Cenários 54

5.3 Resultados e Análise 58

5.3.1 Tempos de Resposta 58

5.3.2 Consumo de Recursos 62

5.3.3 *Safety* 64

5.4 Ameaças à Validade 64

5.5 Resumo 66

6 Conclusão **67**

6.1 Limitações 68

6.2 Trabalhos Relacionados 68

6.3 Trabalhos Futuros 71

Referências **72**

Lista de Figuras

1.1 Indivíduo sob monitoramento por uma rede de sensores do corpo humano . 2

2.1 Desenvolvimento de software baseado em engenharia de domínio [Czarnecki e Eisenecker, 2000] 6
2.2 Ciclo de desenvolvimento para suportar atividades de LPSD [Capilla et al., 2014] 8
2.3 Modelo conceitual de LPSD utilizando MAPE [Bencomo et al., 2012] . . . 9
2.4 Modelo de uma rede de sensores do corpo humano [Fernandes, 2012] 11
2.5 Modelo de *features* do aplicativo Monitor [Fernandes, 2012] 11

4.1 Processo de desenvolvimento MEIOq 24
4.2 Modelo de *features* do aplicativo Monitor estendido 24
4.3 Arquitetura original do aplicativo Monitor 26
4.4 Agrupamento de noções semânticas do domínio 26
4.5 Atividades de refinamento da DSL 28
4.6 Modelo da sintaxe abstrata da linguagem 29
4.7 Exemplo de diagrama de máquina de estados da UML 31
4.8 Diagrama de distribuição proposto para flexibilização de LPSDs 39
4.9 Modelo de classes para realização do laço principal de uma VM 42
4.10 Modelo de classes das instruções da máquina virtual 42
4.11 Arquitetura completa do sistema Monitor 46
4.12 Tela inicial atualizada do aplicativo Monitor 47
4.13 Tela de entrada de dados do especialista no domínio 48
4.14 Processo de registro da máquina de estados para receber atualização 49
4.15 Processo de atualização da máquina de estados 50

5.1 Comparação de tempos de execução em dois ciclos de carga e descarga . . 60
5.2 Gráfico de análise dos tempos de atualização 61
5.3 Tempo de ocupação do processador pelo sistema sem coleta de dados (a) e com coleta de dados (b) 62

5.4 Gráfico de análise dos tempos de execução em modo avião 63

Lista de Tabelas

2.1 Tipo de dados das *features* de informação 12

5.1 Definição dos objetivos de avaliação. 53
5.2 Distribuição de configurações por objetivo de qualidade. 57
5.3 Faixas de objetivo de qualidade válidos e inválidos. 57
5.4 Sumarização dos dados. 58
5.5 Tempos para exaustão da bateria 63

Lista de Listagens

2.1 Laço principal típico de uma VM 16
4.1 Exemplo de código da máquina de estados 32
4.2 Gramática da DSL definida usando notação BNF 33
4.3 Definição da gramática usando o ANTLR 34
4.4 Abstração do conjunto de instruções gerada para uma RangeDeclaration . 44
4.5 Série de instruções gerada para uma regra de fusão 44
5.1 Código da máquina de estados usada na avaliação 55

Capítulo 1

Introdução

Em um mundo em constante mudança, o desenvolvimento de software tem, cada vez mais frequentemente, que lidar com alterações nas necessidades dos usuários e se adaptar a cenários de uso não previstos no desenvolvimento inicial de software [Sommerville, 2010]. Tal frequência torna cada vez mais importante a demanda por maior capacidade de arquiteturas de software para comportar mudanças, muitas das quais devem ser realizadas em tempo de execução [Magee e Kramer, 1996]. Para atender esta demanda, surgiram abordagens de desenvolvimento como Linhas de Produtos de Software Dinâmicas (LPSDs) na tentativa de resolver os desafios referentes à dinamicidade de tais sistemas.

Devido à impossibilidade de prever todas as funcionalidade e variantes necessárias a uma linha de produtos de software (LPS), é realizada uma extensão deste modelo com a habilidade de derivar automaticamente mudanças nos requisitos, reconhecidos através da monitoramento do contexto, e automaticamente reconfigurar a aplicação em execução [Hallsteinsen et al., 2006]. Assim, LPSDs podem ser usadas para a construção de sistemas de software capazes de se adaptar a flutuações nas necessidades do usuário e restrições em recursos disponíveis [Hallsteinsen et al., 2008].

Dentre cenários possíveis de uso de LPSDs, podemos citar sistemas para controle automotivo [Shokry e Babar, 2008], sistemas de gerenciamento de relacionamento com o cliente (CRM) [Wolfinger et al., 2008], e redes de sensores sem fio [Ortiz et al., 2012].

1.1 Definição do Problema

Para motivar este trabalho, considerou-se o exemplo de uma aplicação para monitoramento de sinais vitais de um indivíduo através de uma rede de sensores do corpo humano (Figura 1.1). Ao detectar alterações nos sinais vitais recebidos, uma aplicação pode alterar sua configuração para atender a um novo objetivo de qualidade, dentre um conjunto previamente definido de objetivos de qualidade. Por exemplo, se o indivíduo começar a

passar mal, a aplicação passa para um estado de maior risco em que a confiabilidade da aplicação precisa ser maior.

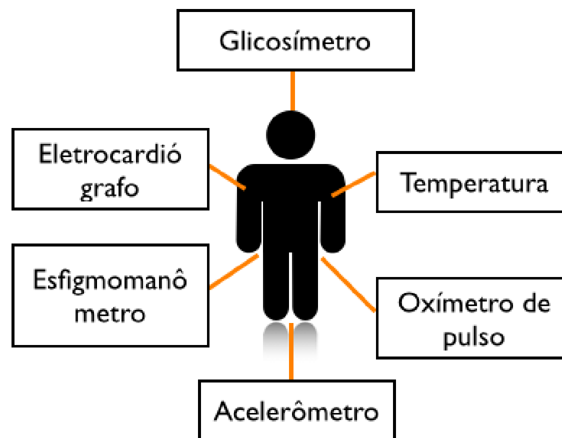

Figura 1.1: Indivíduo sob monitoramento por uma rede de sensores do corpo humano

Entretanto, o conhecimento médico não é estático e está sujeito a alterações que precisam ser consideradas nesta LPSD. Além disso, os próprios indivíduos monitorados tem condições de saúde diferentes uns dos outros, ou seja, certos valores em sinais vitais que são considerados normais para uma parte da população podem representar algum risco de saúde para outra parte da população. Estas características individuais também estão sujeitas a alterações ao longo do tempo tais como o avanço da idade ou a aquisição de doenças.

Através de estudos do estado da arte sobre LPSDs baseadas em objetivos (vide Seção 2.1.2), verificamos que o foco do estudo de LPSDs está em adição de novas *features* e alteração estrutural da LPSD para permitir a criação de novos pontos de variação em tempo de execução, deixando a questão da evolução dos objetivos da LPSD em aberto. Dessa forma, acreditamos que o exemplo citado no início desta seção evidencia a necessidade de estudos a cerca do tema.

1.2 Solução Proposta

Neste contexto, este trabalho se propõe a investigar a possibilidade de criação de mecanismos em LPSDs que permitam a alteração ou evolução dos seus objetivos em tempo de execução, bem como a capacidade de definição de objetivos individualizados para cada instância da LPSD. Isto corresponde ao tratamento de variabilidade no tempo e no espaço descrito por Krueger [2002].

Para resolver o problema da evolução e individualização de objetivos em LPSDs, buscamos desenvolver um **M**odelo para a **E**volução **I**ndividual de **O**bjetivos em LPSDs cientes

de **q**ualidade (**MEIOq**). Com essa abordagem, buscamos permitir a evolução de objetivos de uma LPSD especificada através de uma linguagem específica de domínio.

Para realizar este trabalho, foi elaborado um modelo de arquitetura complementar à arquitetura da LPSD, contendo os mecanismos para especificação de novos objetivos pelo especialista no domínio e para distribuição e utilização dos objetivos atualizados para os dispositivos responsáveis pela execução da LPSD propriamente dita, dessa forma tratando a variabilidade no tempo. Além disso, este modelo e mecanismos preveem que cada instância da LPSD em uso pode possuir um objetivo diferente das demais, por isso dizemos que o modelo permite evolução individual dos objetivos, tratando pontos de variabilidade no espaço.

Em particular, adotamos uma máquina virtual para realizar a execução dos objetivos estabelecidos pelo especialista no domínio. Um compilador da linguagem do especialista no domínio para a linguagem da máquina virtual foi acrescentado à arquitetura do modelo proposto.

1.3 Avaliação

Procedemos à avaliação do trabalho em duas etapas. Na primeira etapa, aplicamos o processo de desenvolvimento proposto, como forma de realizar uma prova de conceito, orientada a demonstrar a viabilidade da abordagem, em uma LPSD para uma rede de sensores do corpo humano, esta criada como resultado do trabalho de Fernandes [2012], no qual os objetivos de qualidade dos sensores (que são *features* para esta LPSD) e os objetivos definidos pelo especialista no domínio influenciam a seleção de configurações quando a LPSD toma a decisão de se reconfigurar.

Em seguida, foi realizada uma avaliação através de simulação para verificar a manutenção do comportamento anterior da LPSD dada a nova implementação e determinar o impacto das alterações propostas sobre a LPSD. Neste sentido, foram avaliados o desempenho e o *safety* do sistema e analisados os resultados. O *safety* do sistema é um atributo da dependabilidade e se caracteriza pela ausência de consequências catastróficas para o usuário ou para o ambiente [Avizienis et al., 2004].

1.4 Resumo de Contribuições

De forma resumida, este trabalho apresenta as seguintes contribuições:

- Um método para evoluir objetivos de LPSDs;

- Definição de um modelo para desenvolvimento de LPSDs que devem observar parâmetros de qualidade durante sua reconfiguração;
- Uma linguagem para especificação de estados de saúde de um indivíduo a partir de caracterização de seus sinais vitais.

1.5 Organização do Trabalho

O restante deste trabalho está organizado da seguinte forma. No Capítulo 2 apresentamos conceitos e fundamentos do estado da arte e estado da prática utilizados neste trabalho. O Capítulo 3 detalha o problema que este trabalho se propõe a resolver. O modelo de evolução de objetivos proposto por este trabalho é apresentado no Capítulo 4 em conjunto com a implementação do modelo em uma LPSD para monitoramento de sinais vitais do corpo humano descrito na seção anterior. No Capítulo 5 realizamos uma avaliação do sistema implementado usando o MEIOq para verificar a manutenção do comportamento da LPSD e se existem vantagens na adoção do nosso modelo. Finalmente, no Capítulo 6, apresentamos as considerações finais sobre este trabalho, os trabalhos relacionados a este e possíveis futuros trabalhos de pesquisa.

Capítulo 2

Fundamentação Teórica

Para melhor orientar o leitor, neste capítulo são apresentados os conceitos que serão utilizados ao longo desse trabalho. Primeiro será feita uma breve introdução ao conceito de linhas de produtos de software, na Seção 2.1, em particular linhas de produtos de software dinâmicas, na Seção 2.1.2; em seguida serão apresentados conceitos sobre redes de sensores do corpo humano (Seção 2.2) e, por fim, os conceitos de linguagens específicas de domínio, na Seção 2.3, e de máquinas virtuais, na Seção 2.4, que também serão abordados neste trabalho.

2.1 Linha de Produtos de Software

Linha de produtos de software (LPS) é uma técnica da engenharia de software que visa ao aproveitamento dos artefatos em comum a uma família de produtos de software e gerencia os artefatos variantes para a composição de produtos relacionados buscando o aumento da qualidade do software, e a redução do tempo e dos custos de desenvolvimento e manutenção [Clements e Northrop, 2002]. Svahnberg et al. [2005] define variabilidade de software como "a habilidade de um artefato ou sistema de software ser eficientemente estendido, modificado, customizado ou configurado para uso em um contexto particular".

Com a introdução de variabilidade, decisões de projeto são adiadas para fases posteriores do desenvolvimento do software, e esta não é uma tarefa trivial [Svahnberg et al., 2005]. Czarnecki e Eisenecker [2000] apresentam um modelo para desenvolvimento de linhas de produtos de software baseado na separação entre atividades relacionadas à engenharia de domínio e à engenharia da aplicação; a Figura 2.1 apresenta este modelo de desenvolvimento.

Logo, os artefatos na LPS devem cobrir todos os elementos a partir dos quais a família de produtos é construída e suas respectivas regras de composição. Porém, esclarecer a forma como as várias partes podem ser combinadas é uma tarefa complexa visto que

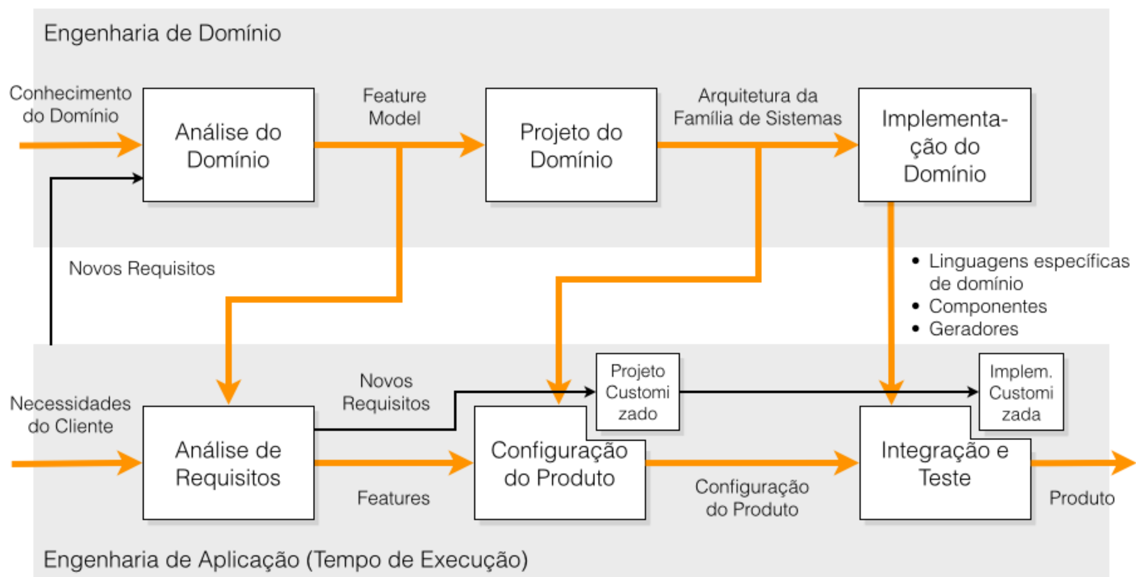

Figura 2.1: Desenvolvimento de software baseado em engenharia de domínio [Czarnecki e Eisenecker, 2000]

existem dependências entre esses elementos e, por consequência, apenas um subconjunto de todas as combinações forma configurações completas e corretas [Anastasopoulos e Gacek, 2001]. A representação formal das variabilidades de uma LPS pode ser feita através de um modelo de *features*, cujas principais relações entre as *features* em determinado ponto de variabilidade são [Czarnecki e Eisenecker, 2000]:

- Obrigatórias: são as *features* comuns a todos os produtos da família;
- Alternativas: um produto só pode ser instanciado com uma dentre as várias *features* alternativas;
- Opcionais: um produto pode ou não apresentar essas *features*.
- OR: um produto pode ser construído com uma ou mais *features* de um conjunto de *features* relacionadas.

2.1.1 Tempo de Ligação

Um modelo de *features* estrutura a variabilidade em uma LPS como um conjunto de variabilidades a serem resolvidas, ao mesmo tempo que captura as relações entre elas [Anastasopoulos e Gacek, 2001]. O momento em que essas variabilidades são resolvidas é chamado tempo de ligação (do inglês *binding time*) e determina quais *features* são selecionadas ou não para um ponto de variabilidade. Essa resolução pode ocorrer em diferentes momentos do ciclo de vida da LPS [Svahnberg et al., 2005]:

- Derivação da arquitetura do produto: a arquitetura do produto pode conter diversos pontos de variação não resolvidos. A resolução desses pontos de variação produz a arquitetura de um produto em particular;
- Compilação: a finalização do código fonte é feita em tempo de compilação, tendo partes removidas por diretivas de compilação ou acrescentadas para adicionar comportamento;
- Ligação: o momento em que a ligação (*linking*) ocorre pode ser forçado pelo compilador ou ocorrer apenas quando o sistema entra em execução. Em alguns casos, o sistema em execução tem liberdade para refazer essas ligações;
- Execução: a resolução de variabilidades em tempo de execução pode ser feita com uma lista aberta ou fechada de variantes. Uma lista fechada não permite que novas variantes sejam adicionadas é LPS, ao contrário da lista aberta. Tais variantes são comumente chamadas de *plug-ins* e podem ser implementadas por terceiros.

Esta classificação, entretanto, não é unanime e pode ser expandida com outros tempos como, por exemplo, tempo de pré-processamento, instalação, distribuição, ou mesmo outros tempos relativos ao ciclo de vida de um produto de software específico [Czarnecki e Eisenecker, 2000].

O tempo de resolução determina quantos produtos ainda podem ser derivados da LPS em determinado tempo considerando as variabilidades já resolvidas e as deixadas em aberto [van Gurp et al., 2001]. O tempo de resolução também influencia o custo final da LPS, pois quanto mais a resolução de uma variabilidade for adiada, mais cara será sua implementação [Svahnberg et al., 2005].

2.1.2 Linha de Produtos de Software Dinâmica

Dada a frequente necessidade de atendimento a alterações dinâmicas nos requisitos dos usuários e ambientes de sistemas, linhas de produtos de software dinâmicas (LPSDs) surgem como uma evolução de LPSs que permite a geração de variantes do produto em tempo de execução [Lee e Kang, 2006; Hallsteinsen et al., 2006; Wang et al., 2006; Hallsteinsen et al., 2012]. O estudo de LPSDs ainda é recente e possui pontos de sobreposição com sistemas auto-adaptativos [Bencomo et al., 2012], porém, ao descrever as adaptações de programas em termos de *features*, é possível criar abstrações dos detalhes da implementação, simplificando a reconfiguração dos programas em execução e facilitando a verificação de consistência das adaptações [Rosenmüller et al., 2011].

Capilla et al. [2014] descreve que características e desafios relacionados a LPSDs fazem com que sua organização seja diferente de LPSs tradicionais, requerendo novas atividades

para suportar variações em tempo de execução em artefatos e produtos, tal como ilustrado pela Figura 2.2.

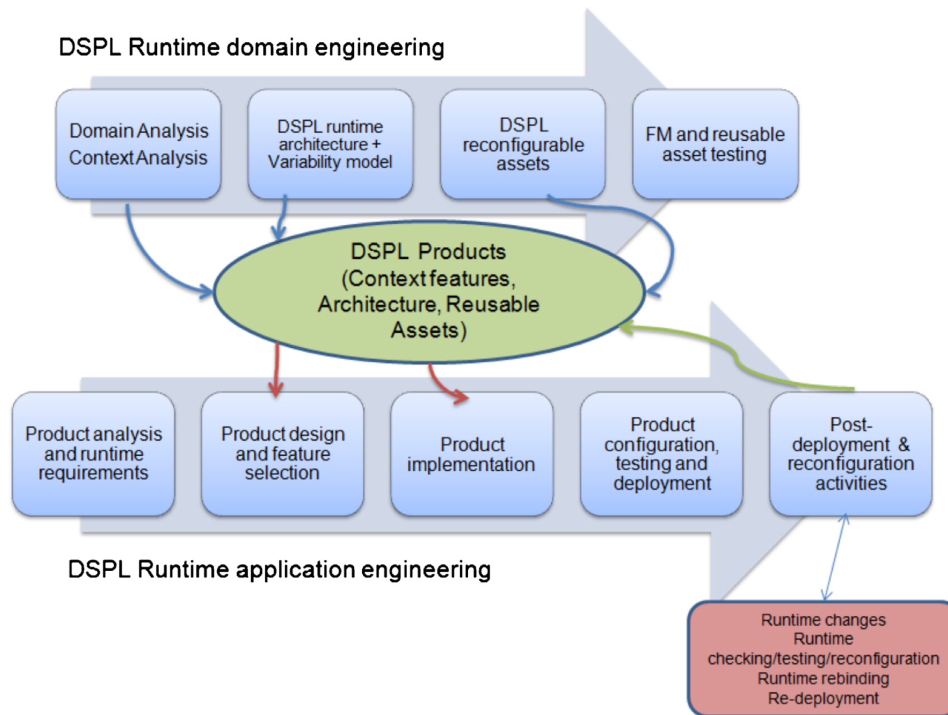

Figura 2.2: Ciclo de desenvolvimento para suportar atividades de LPSD [Capilla et al., 2014]

Enquanto em uma linha de produtos clássica produtos são derivados de uma infraestrutura para um mercado ou indivíduo específico e não sofrem alterações depois de serem distribuídas, em uma LPSD produtos são reconfigurados dinamicamente em tempo de execução, baseado em modelos derivados e representados utilizando práticas clássicas de LPS [Capilla et al., 2014]. A reconfiguração de uma LPSD pode ser disparada por meio de interações com o usuário, o qual deve entender os efeitos da reconfiguração, ou por eventos ocorridos no contexto da aplicação, os quais podem ser mais difíceis de serem reproduzidos durante o desenvolvimento [Cetina et al., 2010].

O modelo MAPE-K (ou simplesmente MAPE), usado na computação autônoma [Kephart e Chess, 2003], cujo nome é derivado das principais atividades em um típico laço de controle em um sistema auto-adaptativo, é comumente associado à implementação de LPSDs. A Figura 2.3 apresenta o modelo conceitual de uma LPSD usando MAPE definido por Bencomo et al. [2012]. Nesta figura, o laço superior equivale às atividades realizadas manualmente pelas pessoas envolvidas com o desenvolvimento de LPSDs para a engenharia de domínio, essencialmente associado à evolução da infraestrutura da LPSD, enquanto o laço inferior é utilizado pela implementação da LPSD para apoio ao processo de reconfiguração dinâmica do produto após uma derivação inicial.

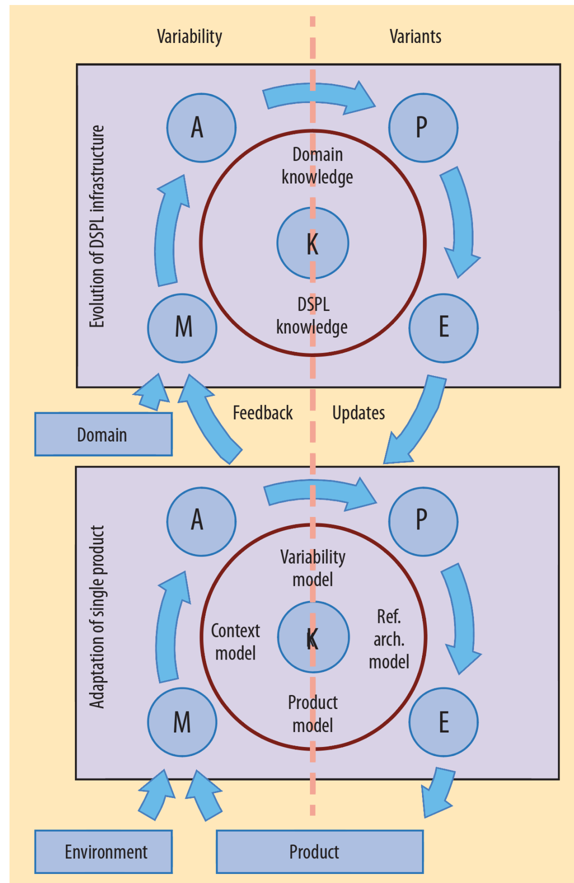

Figura 2.3: Modelo conceitual de LPSD utilizando MAPE [Bencomo et al., 2012]

O laço MAPE é iniciado pela monitoramento de eventos que possam disparar uma necessidade de reconfiguação da LPSD; ao reconhecer um desses eventos, uma atividade de análise é realizada para identificar impactos da mudança sobre os requisitos e restrições do produto; uma vez confirmada a necessidade de reconfiguração, é feito um planejamento para derivar a configuração adequada para a situação; e por fim é executada a reconfiguração necessária [Bencomo et al., 2012]. Operações de validação e verificação em tempo de execução podem ser necessárias após a reconfiguração do produto e antes que ele retorne à sua operação normal [Capilla et al., 2014].

2.2 Rede de Sensores do Corpo Humano

Com avanços no uso de tecnologias sem fio, tornou-se possível a utilização de dispositivos com esta tecnologia para o provimento de serviços de sáude. Um dos conceitos mais promissores é o de redes de sensores do corpo humano (RSCH). Uma RSCH é um sistema composto por sensores utilizados por um indivíduo e que operam de forma autônoma com

o objetivo de coletar e processar dados de sinais vitais de um indivíduo monitorado e comunicar sem fio um ou mais dados fisiológicos ou do ambiente [Hao e Foster, 2008].

Conforme dados sobre o indivíduo são coletados pelos sensores e analisados pelo sistema centralizado, este último pode determinar se o indivíduo encontra-se em risco ou não. Uma situação de mais alto risco pode exigir que mais dados sejam coletados e/ou com maior frequência enquanto uma situação de mais baixo risco pode permitir economizar recursos desativando sensores desnecessários [Fernandes, 2012]. Hao e Foster [2008] elenca os principais requisitos e desafios para sistemas de medição fisiológica.

- *Confiabilidade* de que a informação chegará ao seu destino, a qual depende de aspectos como confiabilidade de comunicação, computação eficiente em cada sensor e programação de software estável;
- *Biocompatibilidade*, pois forma, tamanho e materiais a serem usados em sensores que atuam diretamente sobre o corpo humano são restritos;
- *Portabilidade*, uma vez que os sensores devem ser pequenos e leves para permitir que sejam usados presos ao corpo ou mesmo engolidos;
- *Privacidade e segurança* para evitar questões como roubo de identidade ou captura de dados privados por pessoas não autorizadas; e
- *Uso eficiente de energia* para comunicação de dados e permitir que sensores possam operar com consumo mínimo de energia.

A necessidade de acompanhamento de um indivíduo varia de acordo com o seu estado de saúde e, consequentemente, altera os requisitos exigidos da RSCH. Devido à sua natureza, é necessário que esta reconfiguração seja feita de forma a manter um determinado nível de qualidade de serviço. A variabilidade de sensores disponíveis e necessidade de uma reconfiguração correta tornam a RSCH um forte candidato a implementação como LPSD.

2.2.1 O Aplicativo Monitor

Fernandes [2012] propõe o uso de uma LPSD para reavaliar constantemente a situação de um indivíduo monitorado através de uma RSCH (Figura 2.4), e selecionar uma configuração adequada para um estado de risco determinado. Os requisitos de estados são traduzidos em objetivos de qualidade, utilizados pela solução ao transitar para determinado estado.

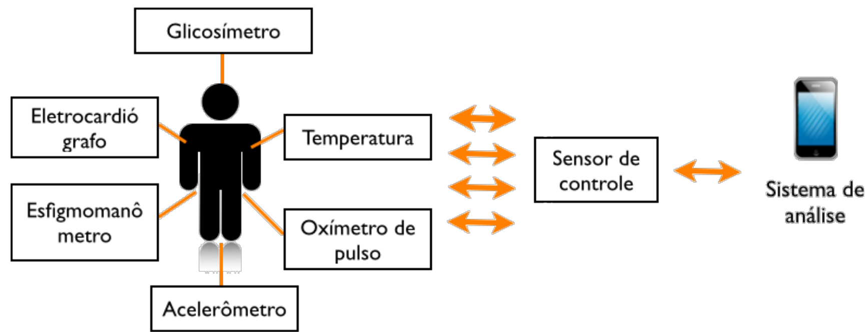

Figura 2.4: Modelo de uma rede de sensores do corpo humano [Fernandes, 2012]

A Figura 2.5 apresenta o modelo de *features* desta LPSD [Fernandes, 2012]. Como podemos observar a partir do modelo de *features*, este apresenta três tipos de *features* por agrupamento semântico:

- *Features de armazenamento* especificam o meio de armazenamento dos dados lidos dos sensores;
- *Features de sensores* representam os sensores que podem ser utilizados pela LPSD;
- *Features de informação* são informações que podem ser extraídas dos sensores;

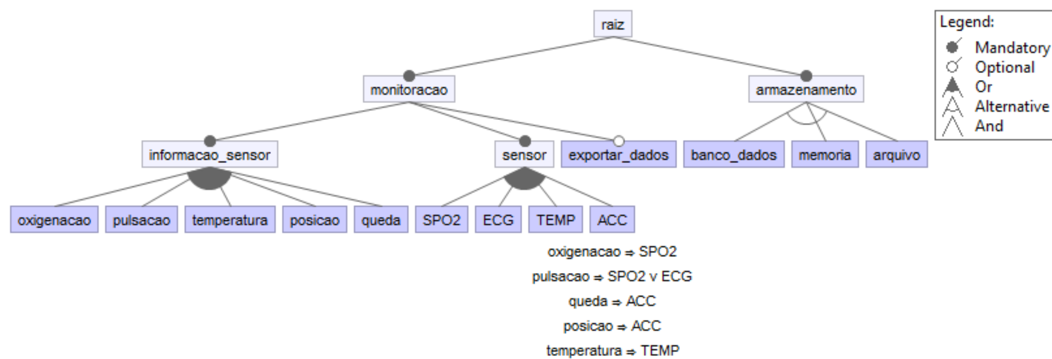

Figura 2.5: Modelo de *features* do aplicativo Monitor [Fernandes, 2012]

Assim, um sensor pode fornecer mais de uma informação por vez, como por exemplo o acelerômetro (*feature* ACC na figura) que pode ser utilizado para determinar a posição do indivíduo (*feature* posicao) e identificar se o indivíduo caiu (*feature* queda). Da mesma forma, uma informação pode ser extraída de mais de um sensor; a informação de pulsação (*feature* pulsacao) pode ser obtida através do eletrocardiograma ou do oxímetro de pulso (*features* ECG e SPO2, respectivamente), porém com atributos de qualidade diferentes; neste caso, a implementação da *feature* deve resolver qual valor será atribuído à informação em dado momento.

Cada *feature* de informação fornece apenas um dado extraído dos sensores. O tipo desse dado pode ser:

- *Numérica*, representando um valor do tipo inteiro ou ponto flutuante;
- *Enumeração* de valores possíveis agrupados em uma lista;
- *Booleana*, apenas sim ou não.

A Tabela 2.1 apresenta os tipos de dados associados a cada *feature* de informação do aplicativo Monitor.

Feature	Tipo de dado
oxigenacao	numérico
pulsacao	numérico
temperatura	numérico
posicao	enumeração
queda	booleana

Tabela 2.1: Tipo de dados das *features* de informação

Para realizar a seleção do estado adequado do indivíduo monitorado, Fernandes [2012] utiliza um autômato determinístico finito na forma de uma máquina de estados. A máquina de estados definida atinge seu objetivo através da análise de dados advindos de sensores usados por um indivíduo sob monitoramento e recebidos pelo aplicativo Monitor através de conexão *Bluetooth*. Os dados são análisados segundo regras predefinidas na máquina de estados, que classifica o conjunto destes dados em três estados possíveis também estáticos: Normal, Risco Moderado e Risco Alto. Alterações nas regras e estados podem ser realizadas somente em tempo de compilação, para expandir ou retrair o número de regras e estados disponíveis, bem como para realizar alterações em suas características.

A solução proposta no trabalho de Fernandes [2012] foi implementada para plataforma Android, realizando a leitura de sinais dos sensores através de *Bluetooth*.

2.3 Linguagens Específicas de Domínio

Ao contrário de linguagens de propósito geral, linguagens específicas de domínio (*domain-specific languages* ou DSL) são linguagens construídas especificamente para um domínio de aplicação em particular, possibilitando desenvolver aplicações completas para aquele domínio de forma rápida e eficiente [Hudak, 1998]. Com o uso de tais linguagens, é possível obter ganhos substanciais de expressividade e facilidade de desenvolvimento pois elas

são mais expressivas para um domínio limitado, e proveem notações e construções mais adequadas em comparação às linguagens gerais [Mernik et al., 2005].

DSLs são linguagens pequenas, oferecendo um conjunto restrito de notações e abstrações e são, geralmente, declarativas. Tais linguagens declarativas também são vistas como linguagens de especificação, mas não deixam de ser tratadas como linguagens de programação. Muitas são suportadas por um compilador capaz de gerar aplicações a partir de códigos escritos na DSL. Outras linguagens são usadas para gerar bibliotecas ou componentes, ao invés de programas completos, ou para geração de documentos ou imagens [van Deursen et al., 2000].

Uma DSL bem projetada deve ser tão simples quanto possível ao mesmo tempo em que é tão poderosa quanto necessário, balanceando os riscos e oportunidades associados à adoção dessa técnica na engenharia de software, dentre os quais, podemos citar [van Deursen et al., 2000]:

- Expressão de soluções no idioma e nível de abstração do domínio do problema;
- Geração de programas concisos, compreensíveis e reutilizáveis para diferentes propósitos;
- Aumento na produtividade, confiabilidade, manutenibilidade e portabilidade;
- Validação e otimização no nível do domínio da aplicação;
- Alto custo no projeto, implementação, manutenção e disseminação da DSL;
- Dificuldade em encontrar o escopo correto da DSL;
- Dificuldade em balancear especificidade do domínio e propósito geral da linguagem; e
- Potencial perda de eficiência comparado ao software codificado à mão.

Uma linguagem pode ser definida utilizando notação (forma de representação dos conceitos) gráfica ou textual. Linguagens com notação gráfica utilizam símbolos visuais que representam conceitos do domínio e permitem ao especialista estabelecer visualmente relações entre esses elementos, porém dependem de ferramentas de edição especializadas para serem trabalhadas. Linguagens com notação textual, por sua vez, permitem representar as mesmas informações, porém utilizando textos estruturados com regras formais.

Linguagens textuais são preferidas por especialistas de domínio por fornecerem uma visão geral melhor dos conceitos descritos na linguagem e pela facilidade para definição por digitação, em contraposição à modelagem gráfica [Green e Petre, 1992].

Outro aspecto importante sobre DSLs tem relação com sua execução. DSLs são executáveis de várias formas e em vários graus, mesmo ao ponto de não serem executáveis; tais linguagens são melhor chamadas de especificações, definições ou descrições. Alguns desses graus de executabilidade são definidos por Mernik et al. [2005]:

- DSLs com semântica de execução bem definida;
- Linguagens de entrada para um gerador de aplicações, comumente declarativas e com menor semântica de execução definida, onde o gerador funciona como compilador para a linguagem;
- DSLs não criadas com o intuito primário de serem executadas, mas úteis para a geração de aplicações; e
- DSLs não criadas para serem executadas, mas que podem se beneficiar de suporte ferramental como editores especializados, *pretty printers*, verificadores de consistência e analisadores.

2.3.1 Processo de Definição

Strembeck e Zdun [2009] realizaram diversas experiências no desenvolvimento de diferentes tipos de DSLs e, a partir destas, sintetizaram um processo de desenvolvimento composto por quatro atividades. Os autores porém relatam que a ordem em que estas atividades devem ser executadas e os passos exatos a serem realizados podem variar significativamente sob a influência de diversos fatores como tamanho do projeto e envolvimento dos *stakeholders*, porém a existência de um processo sistemático é um ótimo guia para o desenvolvimento de DSLs, que também é descrito como um processo iterativo e exploratório na maioria dos casos.

van Deursen et al. [2000] acrescenta que um dos requisitos para desenvolvimento de uma DSL é ter o conhecimento do domínio amadurecido, e por isso DSLs são consideradas a fase final e mais madura de evolução de um *framework* de aplicação. Por sua vez, van Deursen et al. [2000] também acrescenta que o desenvolvimento de uma DSL como composto tipicamente por seis passos, dividido em dois grupos: atividades de análise (1 a 4) e atividades de implementação (5 e 6):

1. Identificar o domínio do problema;
2. Reunir todo conhecimento relevante sobre este domínio;
3. Agrupar este conhecimento em noções semânticas e operações sobre elas;

4. Projetar uma DSL que concisamente descreve aplicações neste domínio;
5. Construir uma biblioteca que implementa as noções semânticas;
6. Projetar e implementar um compilador que traduz programas da DSL em uma sequência de chamadas à biblioteca.

As atividades de análise buscam a obtenção de conhecimento e modelagem sistemática do domínio da aplicação, o qual também é usado para desenvolvimento de famílias de sistemas correlatos, o que por sua vez tem relação com linhas de produto de software [van Deursen et al., 2000].

Pudemos observar a existência de diversas similaridades entre ambas as abordagens e atividades descritas por Strembeck e Zdun [2009] e van Deursen et al. [2000] para o desenvolvimento de DSLs, exceto que o primeiro considera que as atividades de identificação e levantamento do conhecimento do domínio já foram realizadas antes do início do processo, e não fixa o modelo de implementação.

Também pudemos verificar que a definição do modelo básico da linguagem definido por Strembeck e Zdun [2009] descreve um modo de construir noções semânticas a partir de abstrações do domínio, equivalente ao passo 3 descrito por van Deursen et al. [2000]. Por fim, ambas as abordagens descrevem os passos finais do processo de desenvolvimento com a implementação efetiva da DSL e sua integração com a infraestrutura da aplicação.

2.4 Máquinas Virtuais

Máquinas virtuais (VM) são uma forma para implementação de suporte a uma linguagem de programação que reune características de compilação e interpretação para uma arquitetura computacional implementada em software. Além de serem usadas como meio para garantir portabilidade, máquinas virtuais são também um meio para aumentar a velocidade geral da execução de programas, provendo um ponto central para otimização [Craig, 2006] e simplificando o processo de compilação propriamente dito, uma vez que possibilita o uso de conjuntos de instruções mais adequados à linguagem em vista [Wilhelm e Seidl, 2010].

Comumente, programas compilados em instruções de máquina de uma VM são interpretados por um interpretador [Wilhelm e Seidl, 2010], sem obrigatoriedade de relação entre o código-fonte e o código compilado para a VM. Lindholm et al. [2013], por exemplo, cita que a máquina virtual Java não detém qualquer conhecimento sobre a linguagem de programação Java, permitindo que qualquer linguagem cuja funcionalidade possa ser expressa no código reconhecido pela máquina virtual seja executado pela mesma. Outra vantagem do uso de máquinas virtuais citada por Craig [2006] é a portabilidade do

código bem como a execução de linguagens com baixa ou nenhuma conformidade com a arquitetura da plataforma alvo.

Entretanto, a adoção de uma máquina virtual acarreta em custos de desenvolvimento e manutenção que variam de acordo com o tamanho e a complexidade da VM [Craig, 2006]além de introduzirem um *overhead* no processamento de um sistema e serem um ponto adicional para a ocorrência de defeitos.

2.4.1 Implementação da VM

A Listagem 2.1 apresenta um exemplo de laço principal tipicamente encontrado na implementação de uma máquina virtual (adaptado de [Craig, 2006] para a linguagem Java).

```
1  int ip = 0;
2  boolean halt = false;
3  while (!halt) {
4    byte instr = code[ip++];
5    switch (instr) {
6      case ADD:
7        ...
8      case JUMP:
9        ...
10       ip = new_value;
11       ...
12     case HALT:
13       halt = true;
14       ...
15     default:
16       ...
17       halt = true;
18       ..
19   }
20 }
```

Listagem 2.1: Laço principal típico de uma VM

Nesta listagem, cada constante na instrução `switch` representa uma operação da máquina virtual. Durante a execução de um programa na máquina virtual, o código de uma instrução é recuperado do código e tratado pela instrução *switch*; os detalhes da implementação de cada operação são específicos da implementação de cada máquina virtual em particular. O controle da instrução seguinte a ser executada pela máquina virtual é dado por um registrador chamado ponteiro de instrução (*instruction pointer*, IP), cujo valor comumente pode ser alterado por uma instrução para criar desvios (condicionais

ou não) e laços de repetição. Ao retornar para o início do bloco de repetição `while` o valor do ponteiro de instrução deve apontar para a próxima instrução a ser executada.

Outra forma comum de implementação de máquinas virtuais é a tradução do programa para outra linguagem ou outra máquina virtual. Também é possível realizar a tradução do programa na máquina virtual para instruções de um processador real, chamado *código nativo*, quando o programa é referenciado ou chamado a primeira vez. Esta abordagem, conhecida como compilação *Just In Time* (JIT), tem sido empregada pelas principais plataformas baseadas em VM no mercado, mas impõe a sobrecarga de recompilar todas as unidades do programa sem distinção por frequência de chamadas ou custo de execução [Craig, 2006]. O objetivo da compilação JIT é combinar eficiência com portabilidade [Wilhelm e Seidl, 2010].

2.4.2 Pequenas Plataformas

Craig [2006] descreve em seu trabalho uma área de estudo de interesse relacionada a máquinas virtuais a qual nomeia "pequenas plataformas". Ele caracteriza pequenas plataformas pela existência de armazenamento principal pequeno, armazenamento secundário mínimo ou inexistente, processadores de baixo desempenho (apesar de reconhecer mudanças neste aspecto) e limitações no consumo de energia, tais como *wearables*, tablets, celulares e sistemas embarcados. O laço principal desse tipo de máquina virtual tende a ser relativamente pequeno, porém o código de suporte pode ser proporcinalmente maior [Craig, 2006].

Código compacto é o principal benefício de plataformas pequenas. Por exemplo, sequências comuns de instruções podem ser agrupadas em uma única instrução da máquina virtual, abreviando o código. Da mesma forma, não é necessário que a máquina virtual suporte todo tipo de instruções, podendo, por exemplo, omitir instruções de ponto flutuante bem como diversas instruções de baixo nível de acesso à pilha em favor de instruções de mais alto nível [Craig, 2006].

2.5 Resumo

Neste capítulo foram apresentados os principais fundamentos teóricos utilizados ao longo do restante deste trabalho. No próximo capítulo, serão detalhados os problemas que buscamos resolver com a abordagem proposta.

Capítulo 3

Modelagem do Problema

Para melhor caracterizar este trabalho, apresentamos neste capítulo uma descrição da sua motivação e dos cenários que buscamos cobrir com a adoção do MEIOq. O conteúdo deste capítulo está organizado da seguinte forma. Na Seção 3.1 é apresentada a motivação para este trabalho; na Seção 3.2, são apresentados alguns cenários de evolução de objetivos de LPSDs; e, por fim, os requisitos que devem ser atingidos com este trabalho são elencados na Seção 3.3.

3.1 Motivação

A necessidade de crescente de adaptação de sistemas durante sua execução com base na mudança de contextos ou requisitos de uso tem se tornado cada vez mais comum [Lee e Kang, 2006; Hallsteinsen et al., 2006; Wang et al., 2006]. Para atender a essa necessidade, surgiu o conceito de linha de produto de software dinâmicas (LPSD), que é uma extensão de linhas de produto de software (LPS) com a haiblidade de reconfiguração autônoma em tempo de execução, substituindo componentes de acordo com as necessidades do contexto para melhor atingir os objetivos da linha de produtos. Esta característica, apesar de conferir um alto nível de automação, reduz a capacidade de evolução da linha de produtos dada a necessidade conhecimento prévio dos possíveis pontos de variação e respectivas variantes.

Sistemas adaptativos lidam com o problema da evolução de necessidades e do ambiente de forma mais automática em tempo de execução, porém as abordagens propostas exploram mecanismos de aprendizado de máquina, o que corresponderia a automatizar parte do processo de engenharia do domínio [Bencomo et al., 2012].

Entretanto, é necessário um nível maior de interferência de um especialista no domínio, bem como de confiabilidade, em sistemas que lidam com a vida humana. Além disso, os objetivos do software ou o conhecimento sobre o domínio podem evoluir ao longo do

tempo e tais alterações também devem ser suportadas. Bencomo et al. [2008] descreve que a capacidade de adaptação de mais alto nível de uma linha de produtos requer maior envolvimento humano no processo de adaptação o que pode apresentar riscos à integridade do sistema, deixando o sistema em um estado inseguro.

Neste contexto, podemos verificar a presença de uma questão em aberto sobre a possibilidade de evolução de objetivos em LPSDs (vide Seção 2.1.2). Até onde podemos perceber através de estudo do estado da arte (vide Seções 2.1.2 e 6.2), a maior parte dos trabalhos relacionados com a evolução de LPSs se preocupa somente com a extensão ou substituição do modelo de *features*.

3.2 Cenários

Para melhor caracterizarmos o problema que será alvo deste trabalho, tomaremos como exemplo o aplicativo Monitor criado por Fernandes [2012] para avaliar a reconfiguração de uma LPSD de acordo com objetivos de qualidade previamente estabelecidos e outras informações sobre o contexto da aplicação. Sobre este sistema, apresentamos os seguintes cenários que não foram previstos no desenvolvimento inicial.

Cada pessoa possui características próprias que a diferem das demais; o mesmo se aplica aos sinais vitais em um indivíduo. Valores de características como pressão arterial ou batimentos cardíacos considerados normais podem variar entre indivíduos diferentes (variação no espaço) e até mesmo em um mesmo indivíduo ao longo de sua vida (variação no tempo). Adicionalmente o próprio conhecimento médico pode ser revisto e alterado, criando condições especiais que não foram previstas no desenvolvimento inicial do sistema de monitoramento.

Logo, a evolução de objetivos da LPSD deve ser realizada por um especialista no domínio da aplicação e de forma controlada, de modo a não causar riscos aos indivíduos monitorados. Neste cenário, também se faz necessária a capacidade de redefinir objetivos individualmente, ou seja, de especificar objetivos diferentes para cada instância da LPSD.

Por fim, também é importante considerar que cada instância desta LPSD está em execução em um dispositivo móvel, o qual pode não estar na presença do especialista no domínio durante a atividade de evolução dos objetivos da LPSD, por exemplo, em casos de evolução do conhecimento médico associado ou de atualização da especificação dos objetivos ser realizada por uma terceira pessoa.

3.3 Requisitos

Com base nos cenários definidos anteriormente, identificamos alguns requisitos a serem atendidos para permitir a evolução dos objetivos de LPSDs em geral, os quais elencamos a seguir.

R.1 *Livre expressão de objetivos*, ou seja, é desejável que o especialista no domínio seja capaz de expressar os objetivos da LPSD da forma mais ampla possível a partir de um conjunto de regras relacionadas ao domínio da aplicação e não simplesmente através de expansão do modelo de *features*;

R.2 *Independência de localização do dispositivo* onde a LPSD está em execução. Isto dá ao especialista no domínio a capacidade de atualização das regras da LPSD independente da localização geográfica do especialista ou do dispositivo onde a LPSD está em execução, por isso não deve ser necessário que a LPSD pare sua operação normal para substituição dos objetivos;

R.3 *Otimização de desempenho* dado o risco associado ao tempo para reconfiguração de uma LPSD que deve atender objetivos específicos de qualidade, é necessário que a evolução da mesma não traga prejuízo perceptível ao desempenho da LPSD;

R.4 *Manutenção em um estado consistente*, evitando falhas que possam levar à interrupção das suas funções normais por falhas na atualização da mesma ou pela existência de definições inconsistentes ou incompletas;

R.5 *Garantia de atualização* da LPSD imediatamente ou, em caso de falha temporária de conexão que a impeça, no momento mais próximo possível;

R.6 *Controle de alteração* da LPSD de modo a garantir a impossibilidade de alteração da LPSD por um indivíduo não-capacitado diretamente no dispositivo onde esta se encontre.

R.7 *Atualização de configurações válidas*, garantindo que a LPSD será capaz de encontrar uma configuração válida sempre que os objetivos determinem sua reconfiguração;

R.8 *Individualidade de perfis*, permitindo a redefinição da LPSD para diferentes propósitos de acordo com as necessidades identificadas pelo especialista no domínio, bem como a adequação da LPSD a variações no perfil monitorado ao longo do tempo.

Além dos requisitos para a evolução de LPSDs, ainda encontramos no nosso cenário de avaliação outras necessidades e condições que são específicas do sistema Monitor e que não são atendidas por ele atualmente. Podemos também verificar que a implementação

atual do aplicativo Monitor pode ser adaptada para contemplar os esses novos requisitos e que estes podem ser acrescentados ao modelo proposto neste trabalho.

R.9 *Número variável de estados*, permitindo ao especialista no domínio definir os estados que julgar necessário para o atendimento às suas necessidades de monitoramento;

R.10 *Definição de regras compostas* para determinação do estado atual de saúde do indivíduo monitorado com base nos dados de mais de um sensor;

É importante destacar que este último requisito, *R.10*, foi expresso apenas com o intuito de destacar sua importância para o trabalho atual, tendo em vista que já era suportado pelo trabalho anterior. Da mesma forma devemos observar que o Requisito *R.9* podem ser visto como uma instanciação específica do Requisito *R.1* sem o substituir; o requisito específico apenas estende e complementa o requisito geral.

3.4 Resumo

Neste capítulo, identificamos a importância da evolução dos objetivos de linhas de produtos dinâmicas em tempo de execução como uma oportunidade de estudo. Conhecemos o sistema Monitor, uma redes de sensores do corpo humano implementado como uma LPSD que se reconfigura em resposta a valores obtidos da monitoramento dos sinais vitais de um indivíduo e que será usado por este estudo para avaliação da proposta que será apresentada a seguir. Por fim, identificamos requisitos que esta proposta deve atender para a evolução de objetivos de LPSDs em geral bem como outros especificamente desejados para o sistema Monitor.

Capítulo 4

Evolução de Linhas de Produtos de Software Dinâmicas

Assim como ocorre com outros tipos de aplicações, uma linha de produto de software dinâmica (LPSD) é desenvolvida para atender determinados objetivos específicos, os quais podem ser alterados ou corrigidos posteriormente durante a manutenção do sistema. Uma evolução nos objetivos de uma LPSD pode ser decorrente de um novo conhecimento sobre o domínio da linha de produtos, que não estava disponível em um primeiro momento, ou de alguma particularidade que uma instância específica da LPSD deve considerar.

No desenvolvimento tradicional, o tempo necessário para a adaptação da LPSD aos novos objetivos pode ser relativamente alto, dependendo do processo de manutenção que for adotado. Para reduzir o tempo necessário à evolução dos objetivos da LPSD, este trabalho propõe a adoção de um modelo baseado em linguagens específicas de domínio (DSL) que permita a um especialista definir os novos objetivos da LPSD de forma dinâmica e em tempo de execução.

Uma DSL permite criar um conjunto restrito de notações e abstrações que podem ser utilizadas por um especialista no domínio da LPSD para expressar uma solução no mesmo idioma e nível de abstração do domínio do problema, ao mesmo tempo provendo aumento de produtividade, confiabilidade, manutenibilidade, portabilidade e reuso do conhecimento expresso [van Deursen et al., 2000]. Mernik et al. [2005] reconhece o uso de DSLs em linhas de produtos como um padrão de projeto, para facilitar a especificação de uma LPS dado seu compartilhamento de arquitetura e conjunto básico de componentes, e Czarnecki e Eisenecker [2000] mencionam DSLs como um dos produtos possíveis de implementação na engenharia do domínio.

A adoção de uma DSL como parte da solução para os problemas apresentados no início deste trabalho relacionados à evolução dos objetivos de uma LPSD será apresentada em seguida neste capítulo. Inicialmente apresentaremos o processo elaborado para permitir a

evolução de LPSDs na Seção 4.1, detalhando a construção da DSL como parte da solução na Seção 4.2 e a análise dos objetivos de qualidade da LPSD na Seção 4.3; a Seção 4.4 irá apresentar a arquitetura usada para a implementação da solução, que será detalhada na Seção 4.5.

Ao longo do capítulo, detalharemos a aplicação desta proposta usando o aplicativo Monitor [Fernandes, 2012] como forma de exemplificar os conceitos aqui apresentados. Faremos uso do termo "aplicativo Monitor" quando estivermos nos referindo ao aplicativo Android desenvolvido no trabalho supracitado e usaremos o termo "sistema Monitor" para nos referirmos ao resultado da aplicação do corrente trabalho sobre o aplicativo Monitor, devido a se tratar do sistema composto pelo aplicativo móvel adaptado e um servidor de aplicação web, para apoio à atividade de atualização dos objetivos da LPSD e não existente no trabalho anterior. Onde especificado, detalharemos a construção da DSL e da arquitetura do sistema Monitor utilizando diagramas da *Unified Modelling Language* (UML).

4.1 O Processo de Desenvolvimento

Abaixo apresentamos a Figura 4.1, adaptada de Czarnecki e Eisenecker [2000], que apresenta o processo para desenvolvimento proposto por este trabalho, o qual denominamos **MEIOq** (**M**odelo para **E**volução **I**ndividual de **O**bjetivos em LPSDs cientes de **q**ualidade). Em favor da simplificação, nesta figura são apresentadas, dentro das atividades da engenharia de domínio, apenas as sub-atividades acrescentadas pela nossa metodologia proposta; outras sub-atividades serão brevemente descritas a seguir.

O processo de desenvolvimento MEIOq é dividido em duas partes: a primeira relacionada à engenharia de domínio, representa as etapas de análise, projeto e implementação no desenvolvimento da LPSD, e a segunda relacionada à engenharia da aplicação, que ocorre em tempo de execução para a produção de variantes da aplicação de acordo com os objetivos definidos e com o contexto em que a LPSD se encontra. As atividades relacionadas à engenharia do domínio ocorrem de forma estática, ou seja, em tempo de desenvolvimento (ou de compilação), enquanto a engenharia da aplicação ocorre em tempo de de execução através dos mecanismos de reconfiguração dinâmica da LPSD.

Complementar ao desenvolvimento de uma LPSD comum, nosso processo acrescentou as seguintes atividades ou sub-atividades: Modelagem do Contexto e Objetivos, Refinamento da DSL, Análise de Objetivos de Qualidade, e Redefinição de Objetivos.

A análise do domínio é o primeiro passo a ser realizado para a adoção do MEIOq. Esta atividade consiste na análise do conhecimento sobre o domínio da LPSD para a produção de um modelo de *features*, o qual deve apresentar todos os pontos de variação e respectivas

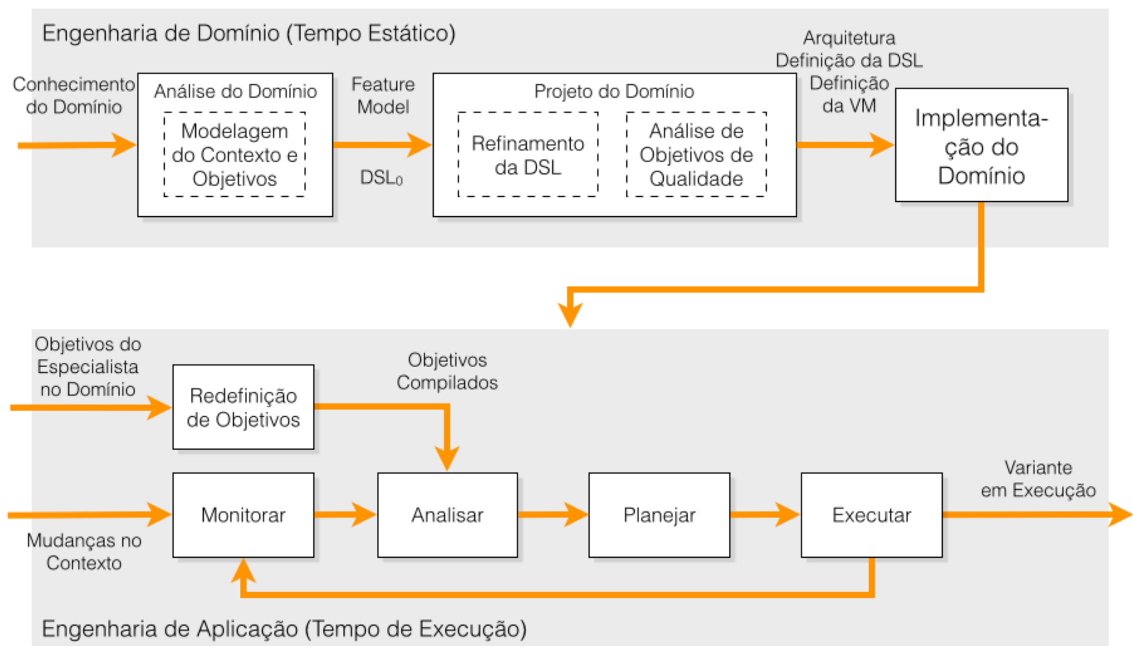

Figura 4.1: Processo de desenvolvimento MEIOq

variantes que estarão disponíveis para a configuração da LPSD em tempo de execução. A Figura 4.2 apresenta o modelo de *features* do sistema Monitor, na qual podemos observar a presença de uma *feature* obrigatória representando os objetivos da LPSD (próxima ao quadro da legenda). Esta é uma *feature* aberta, para a qual não podemos enumerar todas as variantes possíveis, motivo pelo qual estas não são apresentadas abaixo dela.

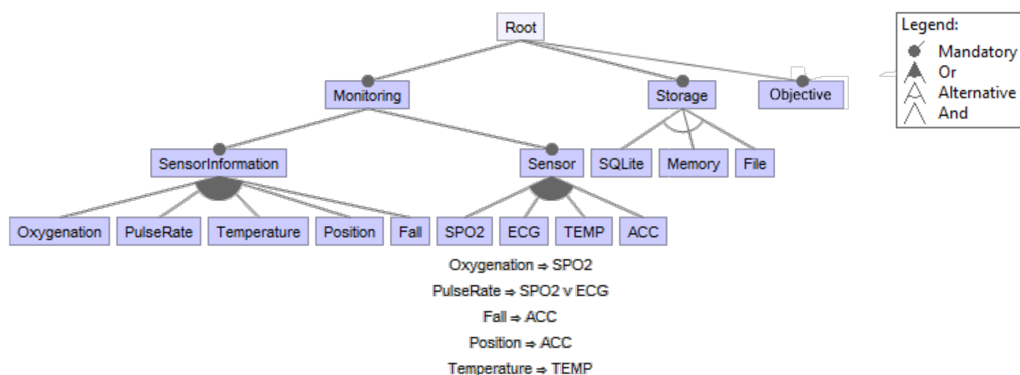

Figura 4.2: Modelo de *features* do aplicativo Monitor estendido

Outro produto da análise de domínio será uma versão preliminar da DSL (DSL_0), fruto da sub-atividade *Modelagem de Contexto e Objetivos*, que será detalhada a seguir na Seção 4.1.1 e refinada durante a atividade de projeto do domínio.

O objetivo do projeto do domínio é fornecer especificações e definições que serão utilizadas na implementação da LPSD. Em adição à especificação da arquitetura da LPSD,

a qual depende de inúmeros fatores que fogem ao escopo deste trabalho, é definida uma arquitetura complementar para permitir o processo de atualização dos objetivos da LPSD com mais baixo acoplamento da arquitetura adotada pela linha de produtos. Também são realizados um refinamento da DSL_0 para a espeficiação da linguagem concreta da DSL e outros elementos de apoio à sua utilização pelo MEIOq, e a análise dos objetivos de qualidade que permearão a seleção de configurações a serem sintetizadas pela LPSD.

Com base nas especificações do projeto, é realizada a implementação da LPSD propriamente dita.

As atividades da engenharia de aplicação são realizadas em tempo de execução, a partir de mudanças nos objetivos da LPSD, definidos por um especialista no domínio, e no contexto. Um ciclo MAPE é reponsável pela captura de eventos que alteram o contexto da LPSD, analisar o novo contexto de acordo com os objetivos definidos pelo especialista no domínio, e planejar e efetuar as mudanças de configuração para se adequar a um novo objetivo de qualidade.

4.1.1 Modelagem de Contexto e Objetivos

O primeiro passo para permitir a flexibilização de objetivos de uma LPSD é identificar o que compõe o contexto utilizado pela aplicação e quais elementos podem ser usado para modelar os objetivos da aplicação.

A rede de sensores do corpo humano a implementada no aplicativo Monitor observa dados biométricos de um indivíduo, através de sensores usados por este, para determinar o estado de risco de saúde do dito indivíduo. Logo, o contexto do aplicativo Monitor é formado pelos dados recebidos destes sensores. A extensão proposta por este trabalho observa que o conhecimento médico associado à determinação do estado de risco do indivíduo, o qual existe de forma independente do sistema, também está sujeito a alterações e precisa ser considerado como parte do contexto no sistema Monitor. Devemos também destacar que o conhecimento médico a cerca de cada indivíduo é diferente e pode evoluir de forma diferente entre indivíduos.

No aplicativo Monitor, este conhecimento médico é representado por uma máquina de estados cujas transições são dadas por regras associadas aos valores percebidos dos sensores utilizados [Fernandes, 2012]. Até o presente momento, é possível realizar a substituição da especificação da máquina de estados, porém apenas em tempo de compilação e não de forma dinâmica em tempo de execução. A Figura 4.3 apresenta arquitetura anterior do aplicativo Monitor; a máquina de estados é um dos componentes do gerenciador de adaptação.

Em seguida, construímos um diagrama de objetos da UML reunindo noções semânticas do domínio relacionados à máquina de estados. Este diagrama foi estruturado em torno

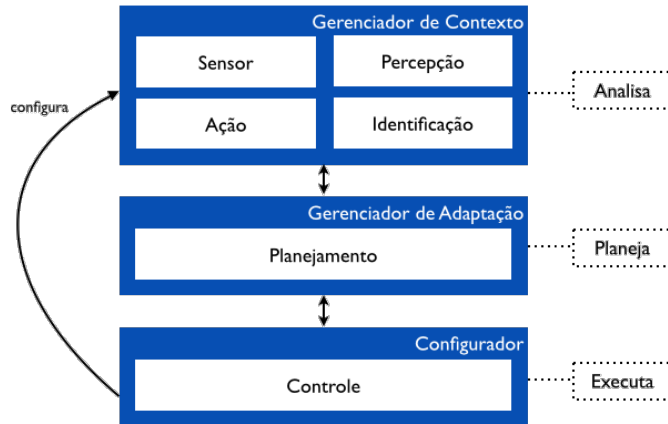

Figura 4.3: Arquitetura original do aplicativo Monitor

de um objeto inicial que serve de raiz da nossa estrutura. Sob este objeto inicial, devem ser acrescidos outros objetos que representam informações agregadas pelo especialista do domínio para orientar o objetivo da LPSD. O resultado final desta composição é uma árvore de objetos que representa uma visão inicial da DSL cujo o nível de detalhamento a ser dado depende das necessidades do sistema e do entendimento dos especialistas no domínio. Este processo pode ser refeito, ou o diagrama resultante reorganizado, diversas vezes até se obter um modelo de objetos que melhor se adeque às necessidades dos especialistas no domínio e à LPSD.

Assim, para o sistema Monitor, foram avaliados os requisitos de evolução da linha de produtos e requisitos específicos do sistema, ambos definidos na Seção 3.3, e analisamos as relações entre informações que poderiam ser utilizadas pelo especialista no domínio para a construção de uma máquina de estados de análise da situação de saúde de um indivíduo monitorado. O diagrama de objetos da Figura 4.4, representa um exemplo das associações de informações identificadas para atender às necessidades do sistema.

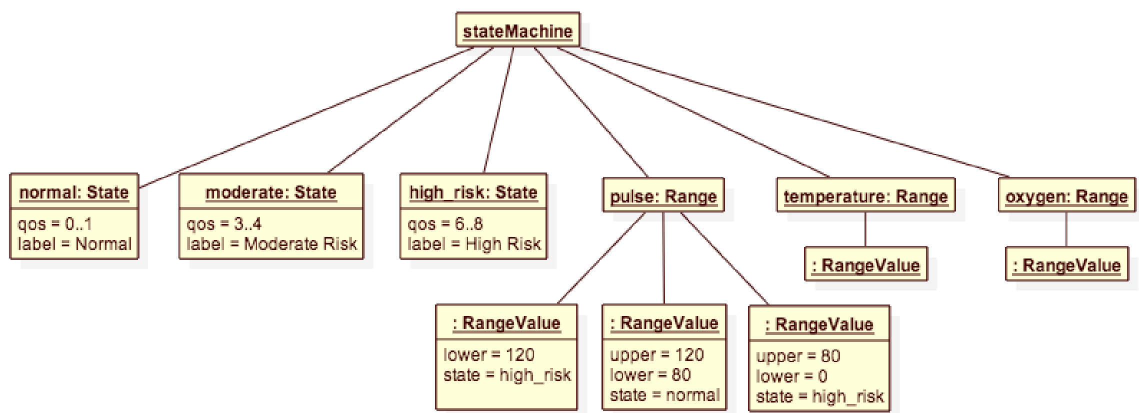

Figura 4.4: Agrupamento de noções semânticas do domínio

Conforme podemos observar por este modelo, uma máquina de estados é representada como um objeto composto por diversos outros que representam dois tipos de elementos: estados (objetos do tipo `:State`) e faixas de valores (objetos do tipo `:Range`). Os estados podem ser associados a diferentes valores que definem suas propriedades, tais como objetivos de qualidade (*qos*) ou etiquetas de texto para exibição (*label*); outras propriedades podem ser acrescentadas no futuro, por exemplo parâmetros para seleção de configurações para o estado. Por sua vez, faixas especificam, de forma simplificada, intervalos de valores para *features* numéricas (vide Seção 2.2.1) que levam a máquina de estados a assumir um determinado estado, também indicado pela respectiva faixa. Os valores nas faixas devem ser ordenados e apenas a faixa mais à esquerda pode omitir o valor de limite superior e a faixa mais à direita pode omitir o limite inferior, ambos indicando que a faixa se extende infinitamente naquela direção. Estas mesmas faixas também podem ter o estado omitido para indicar um estado indefinido, quando houver entendimento que *features* com valores naquelas faixas são erros de leitura e devam ser desconsiderados; as demais faixas devem sempre indicar um estado.

Estas informações são suficientes para a elaboração da linguagem específica do domínio, a qual será alvo da próxima seção.

4.2 Linguagem Específica do Domínio

A partir dos processos discutidos anteriormente na Seção 2.3.1, definimos o processo apresentado na Figura 4.5 para a definição de uma DSL para o MEIOq. Este processo é fortemente baseado em Strembeck e Zdun [2009] com algumas adaptações que serão detalhadas nesta seção.

Strembeck e Zdun [2009] inicia a definição de seu processo apenas indicando que o domínio para a DSL foi previamente escolhido, enquanto van Deursen et al. [2000] explicita a necessidade de reunir o conhecimento relevante sobre o domínio. Ao definirmos uma DSL para uma LPSD, verificamos que este último passo está integrado à engenharia de domínio, tendo sido realizado durante a atividade de análise do domínio descrita anteriormente e pode ser reaproveitado aqui.

Assim, optamos por explicitar a elaboração de uma sintaxe abstrata com base no modelo inicial da DSL, resultante da atividade de modelagem de contexto e objetivos, e que se alinha também com a proposta de van Deursen et al. [2000] para agrupamento de noções semânticas do domínio. Para a definição da sintaxe contreta, mantivemos a linha definida por Strembeck e Zdun [2009] que é mais detalhista e separa as atividades de construção da sintaxe concreta e da definição do comportamento da DSL.

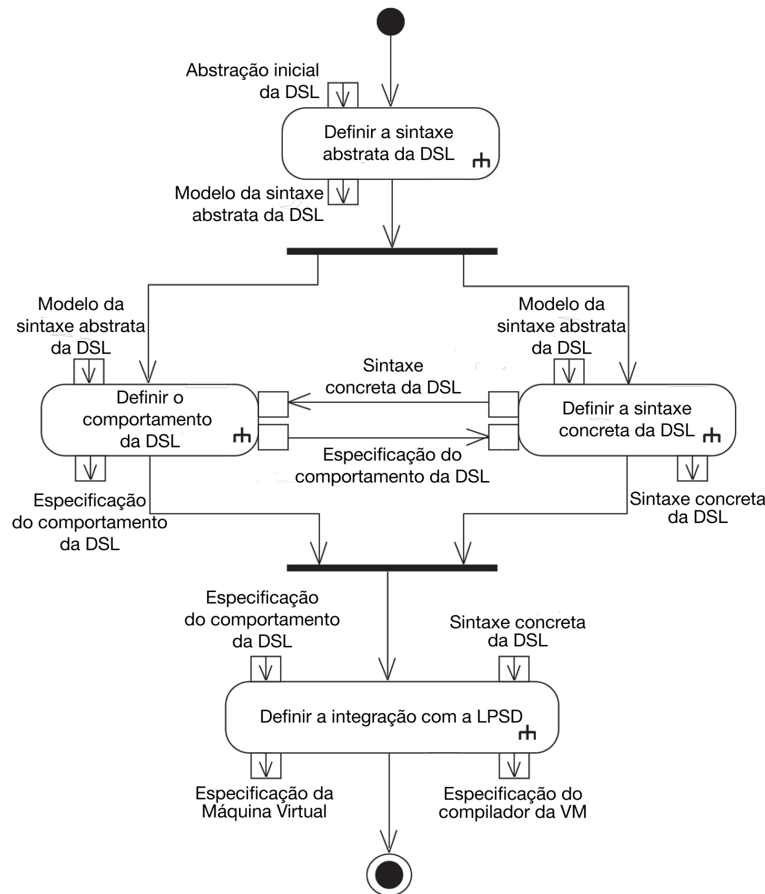

Figura 4.5: Atividades de refinamento da DSL

De modo a atender os Requisitos *R.2* e *R.3*, optamos por integrar à arquitetura da LPSD uma máquina virtual (VM) e um compilador no modelo cliente-servidor com os objetivos de reduzir o tempo de processamento da linguagem pela LPSD e também para permitir a incorporação e flexibilização do comportamento da DSL, permitindo maior reuso de componentes da arquitetura e maior flexiblidade de objetivos posteriormente. Esta decisão também abre a possibilidade de portabilidade futura da LPSD (ou seus componentes) para outras plataformas.

Para dar prosseguimento à definição da DSL do sistema Monitor, procuramos obter a ajuda de profissionais especialistas no domínio médico para definir uma linguagem que melhor representasse o conhecimento relacionado à monitoração de sinais vitais do corpo humano e que pudesse ser facilmente utilizada por estes mesmos profissionais posteriormente para a evolução dos objetivos do sistema.

4.2.1 Definição da Sintaxe Abstrata

No processo de felixibilização dos objetivos da LPSD, o passo seguinte a ser desenvolvido é a definição da sintaxe abstrata para a DSL. Este passo irá permitir a identificação e estruturação das informações que serão usadas pelo especialista no domínio para definir os novos objetivos da LPSD. O resultado deste passo também será usado posteriormente para identificar alterações na arquitetura da LPSD.

Dada a definição inicial da DSL elaborada na análise do domínio, chegamos à definição da sintaxe abstrata da DSL através da abstração dos elementos criados nesta árvore de objetos, e representamos este modelo através de um diagrama de classes da UML. Aqui serão definidas abstrações do diagrama de objetos, cardinalidades, tipos de dados e demais relacionamentos entre as construções da sintaxe abstrata.

Observamos então a Figura 4.6, construída sobre a definição do modelo de objetos visto na Figura 4.4. A sintaxe define dois elementos usados para representar os estados e outro para as faixas de valores. O uso de faixas é opcional e limitado a *features* de valor numérico, com o objetivo de simplificar a leitura do código para esse tipo de *feature*. Outras *features* podem ter valores de tipo lógico (verdadeiro ou falso) ou lista de valores predefinidos (vide Seção 2.2.1) e devem ser definidos diretamente nas declarações de estados.

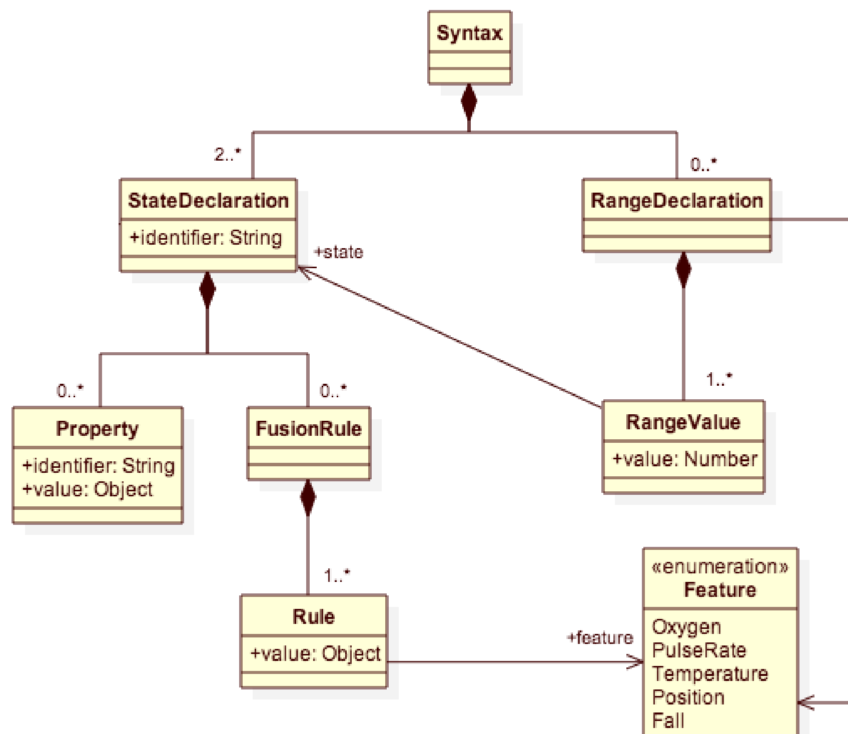

Figura 4.6: Modelo da sintaxe abstrata da linguagem

Optamos por detalhar a seguir a definição de alguns destes elementos na forma textual devido a limitações existentes em outras linguagens, tais como a *Object Constraint Lan-*

guage (OCL), comumente usada em conjunto com diagramas da UML. O suporte a OCL, por exemplo, é implementado de forma diferente por cada ferramenta, o que pode levar a diferentes interpretações do modelo, além de ser sujeito a erros e exigir considerável esforço para sua elaboração [Anastasopoulos, 2013].

Uma declaração de faixas de valores, `RangeDeclaration`, além de referenciar uma *feature*, contém um conjunto de valores (`RangeValue`) que indica um estado e um valor numérico, este último indicando o limite inferior de uma faixa de valores. O estado neste elemento representa o valor que a máquina de estados deve assumir se o valor da *feature* for maior que o valor definido numa `RangeValue`. No primeiro elemento, o estado é opcional e, se ausente, seu valor é usado para indicar o limite superior da faixa seguinte; o estado da *feature* assumirá o valor indefindo caso o valor da *feature* seja maior que o valor definido para a faixa. O último elemento do conjunto também pode ter o valor referente ao seu limite inferior suprimido, indicando que para aquele estado não é testado o respectivo limite. Nos demais elementos do conjunto, ambos os valores são obrigatórios.

O segundo tipo de declaração possível na nossa sintaxe, `StateDeclaration`, representa um estado possível que a máquina de estados poderá assumir e as propriedades (`Property`) deste estado. Estas propriedades podem ser usadas para exibir informações para o indivíduo monitorado ou para adicionar instruções para a seleção de configurações possíveis de um estado, por exemplo. A única propriedade obrigatória é o objetivo de qualidade do estado, o qual é definido por uma faixa de valores numérica e usado como base para a seleção de configurações que podem ser usadas pela linha de produtos quando aquele estado for selecionado. Não há limite conceitual na quantidade de estados que podem ser declaradas em uma única especificação de máquina de estados (Requisito *R.9*).

As regras adicionais para seleção de um estado, ou regras de fusão (`FusionRule`), são avaliadas após as regras nas definições de faixas na ordem em que forem declaradas. Cada regra de fusão é composta por um ou mais de pares de regras simples (`Rule`) contendo cada uma *features* e um valor compatível com o tipo da *feature* indicada. Caso uma regra possua valores para mais de uma *feature*, as avaliações são combinadas com uma operação AND lógica. Esta é a única forma de avaliar *features* dos tipos lógico e enumeração. *Features* de tipo numérico podem ser avaliadas utilizando uma faixa de valores e um estado, este avaliado contra o valor resultante da avaliação de uma declaração de faixa, se esta for usada.

Foi definido um novo tipo de dado, `BigRange`, para representar faixas de valores numéricos no objetivo de qualidade das definições de estados. Este tipo pode ter, opcionalmente, um de seus seus limites inferior e superior indefinido indicando que não existe aquele limite. Esta opcionalidade não é importante para o objetivo de qualidade, cuja faixa de valores válidos varia entre zero (0) e dez (10), porém o mesmo tipo de dado é

utilizado para a definição de faixas de valores em propriedades e pares de regras.

4.2.2 Definição da Sintaxe Concreta

Com base na definição da sintaxe abstrata, podemos definir a sintaxe concreta da DSL, que é a forma como as estruturas definindas para a sintaxe abstrata serão construídas pelo especialista no domínio para refletir os objetivos desejados para a LPSD. A definição da sintaxe concreta irá produzir uma gramática, ou seja, um conjunto de regras usado para garantir que a definição de objetivos da LPSD segue uma estrutura bem definida e não contém elementos que não sejam reconhecidos pelo sistema. Uma gramática pode ser usada para especificar a sintaxe concreta para uma linguagem, tenha ela notação gráfica ou textual.

Como citamos anteriormente, o propósito de uma DSL é permitir que o especialista no domínio expresse uma solução no mesmo idioma e nível de abstração do problema [van Deursen et al., 2000]. Para a DSL do sistema Monitor, poderiamos optar tanto por uma notação gráfica como textual. Uma opção de abordagem para esta DSL com notação gráfica é o aproveitamento de uma linguagem existente, um dos padrões de projeto descritos por Mernik et al. [2005], sobre a UML, neste caso especializando-se no diagrama de máquina de estados da UML (Figura 4.7) extendendo-o para comportar as as construções da sintaxe abstrata não suportadas pela UML (objetivo de qualidade, propriedades e faixas).

Figura 4.7: Exemplo de diagrama de máquina de estados da UML

O uso de uma DSL gráfica implicaria o desenvolvimento de ferramentas para suporte ao desenvolvimento visual das construções da linguagem e seria muito benéfica para os especialistas no domínio. Entretanto, optamos por definir a sintaxe concreta da DSL usando uma notação textual, dada a preferência de especialistas de domínio [Green e Petre, 1992] e devido ao limitado apoio recebido de especialistas no domínio da LPSD para este trabalho.

Assim, analisamos elementos de sintaxe de outras linguagens existentes para definir a forma concreta da sintaxe com a qual trabalharíamos e esboçamos um exemplo de aplicação da linguagem ao modelo de objetos da Figura 4.4, cujo resultado pode ser observado na Listagem 4.1. Esta apresenta o modelo construído na Figura 4.4, com maior riqueza de detalhes não comportados por aquele diagrama, na sintaxe proposta.

```
// State definitions /////////////////////////////////////////////////////

normal state(..1):
  label: "Normal",
  position LAYING + pulse 60..80;

moderate state(5..6.1):
  label: "Moderate Risk";

high state(8..):
  label: "High Risk",
  oxygen moderate + fall yes;



// Feature scales ////////////////////////////////////////////////////////

oxygen range():
  100 > normal > 94 > moderate > 90 > high > 0;

pulse range():
  high > 120 > normal > 80 > high > 0;

temperature range():
  50 > high > 38 > moderate > 37 > normal > 35 > moderate > 30 > high;
```

Listagem 4.1: Exemplo de código da máquina de estados

A capacidade de definir um número variável de estados é um requisito específico desejado para a evolução desta máquina de estados, tendo em vista que a máquina de estados original era capaz de tratar somente três estados predefinidos. Para cada estado, determinamos que algumas propriedades poderiam ser definidas pelo especialista no domínio, tais como uma etiqueta de texto, para indicar visualmente o estado atual selecionado, e o objetivo de qualidade do estado, usado para determinar o conjunto de configurações válidas para dado estado. Outras propriedades poderiam ser adicionadas posteriormente, podendo ser inclusive dependentes do valor de outras propriedades.

Com a definição deste exemplo, partimos para a definição da sintaxe concreta da linguagem e implementação do *parser* da linguagem, que será responsável por transformar o código textual da linguagem em uma árvore de objetos semelhante à da Figura 4.4. A Listagem 4.2 apresenta a sintaxe concreta (também chamada de gramática) da linguagem elaborada em notação BNF, a qual foi posteriormente adaptada para uso com a ferramenta

ANTLR 4[1], devido à sua capacidade de geração de *parsers* simplificados e que melhor se adequavam ao uso na aplicação *web*.

```
1  <start>          ::= <decl> | <decl> <start>
2  <decl>           ::= <state-decl> | <range-decl>
3  <state-decl>     ::= <spaces-opt> <identifier> <spaces> 'state' <spaces-opt>
       '(' <spaces-opt> <range> <spaces-opt> ')' <spaces-opt> ':' <line-end> <
      state-body> <line-end> ';' <line-end>
4  <state-body>     ::= <state-member> | <state-member> <spaces-opt> ',' <line-
      end> <state-body>
5  <state-member>   ::= <spaces-opt> <property> | <spaces-opt> <fusion-rule>
6  <property>       ::= <identifier> ':' <spaces> <value>
7  <fusion-rule>    ::= <rule> | <rule> <spaces-opt> '+' <spaces-opt> <fusion-
      rule>
8  <rule>           ::= <identifier> <spaces> <value>
9  <range-decl>     ::= <spaces-opt> <identifier> <spaces> 'range' <spaces-opt>
       '(' <spaces-opt> ')' <spaces-opt> ':' <line-end> <range-body> <line-end
      > ';' <line-end>
10 <range-body>     ::= <spaces-opt> <range-id-opt> <range-serie> <range-num-
      opt>
11 <range-id-opt>   ::= <identifier> <spaces-opt> '>' <spaces-opt> | ''
12 <range-num-opt>  ::= <spaces-opt> '>' <spaces-opt> <number>
13 <range-serie>    ::= <range-base> | <range-base> <spaces-opt> <range-serie>
14 <range-base>     ::= <number> <spaces-opt> '>' <spaces-opt> <identifier>
15 <value>          ::= <bool> | <string> | <range> | <number> | <identifier>
16 <range>          ::= <number> '..' <number> | <number> '..' | '..' <number>
17 <number>         ::= <digits> '.' <digits> | <digits> | '.' <digits>
18 <string>         ::= '"' <text> '"'
19 <bool>           ::= 'yes' | 'no'
20 <comment>        ::= '//' <text> | '';
21 <spaces>         ::= ' ' <spaces-opt>
22 <spaces-opt>     ::= ' ' <spaces-opt> | ''
23 <line-end>       ::= <spaces-opt> <comment> <EOL> | <line-end> <line-end>
24 <digits>         ::= <DIGIT> | <DIGIT> <digits>
```

Listagem 4.2: Gramática da DSL definida usando notação BNF

O ANTLR gera *parsers* para a linguagem Java, o que também orientou nossa escolha para a implementação do servidor, como discutiremos mais adiante. A Listagem 4.3 a seguir apresenta a definição da gramática na linguagem usada pela ferramenta ANTLR. Nesta definição, a primeira linha funciona como cabeçalho enquanto as demais declaram, cada uma, um elemento terminal (letras maiúsculas) ou não terminal (letras minúsculas) da linguagem.

[1]http://www.antlr.org/

```
grammar Monitor;

start:        decl*;
decl:         stateDecl | rangeDecl;
stateDecl:    identifier 'state' '(' range ')' ':' stateBody ';';
stateBody:    stateMember (',' stateMember)*;
stateMember: property | fusionRule;
property:     identifier ':' (enumerate | value);
fusionRule:   identifier (value | identifier) ('+' fusionRule)?;
rangeDecl:    identifier 'range' '(' ')' ':' rangeBody ';';
rangeBody:    (identifier '>')? rangeBase ('>' number)?;
rangeBase:    number '>' identifier ('>' rangeBase)?;
value:        bool | string | range | number;
identifier:   IDENTIFIER;
range:        (number '..' number?) | ('..' number);
number:       FLOAT | FRACTION;
string:       STRING;
bool:         'yes' | 'no';
enumerate:    ENUM;
ENUM:         [A-Z][A-Z0-9]*;
IDENTIFIER:   [a-z][a-z0-9_]*;
STRING:       '"' ~["\r\n]*? '"';
FLOAT:        DIGIT+ FRACTION?;
COMMENT:      '//' ~[\r\n]* -> channel(HIDDEN);

fragment FRACTION: '.' DIGIT+;
fragment DIGIT:    [0-9];
```

Listagem 4.3: Definição da gramática usando o ANTLR

É importante observar que esta definição não produz uma tradução literal para o diagrama de classes da Figura 4.6, sendo necessário um passo adicional para criar esta correlação, mas é possível trabalhar diretamente com a hierarquia de classes e objetos criada pelo ANTLR, pelo que optamos em virtude do desempenho da avaliação do programa escrito na linguagem e da atualização da máquina de estados. Para permitir a evolução do projeto com a adição de novas *features*, optamos por deixar a sintaxe da linguagem livre para que pudéssemos tratar os tipos de cada *feature* na validação semântica.

Com a definição da gramática da DSL, a ferramenta ANTLR é capaz de gerar algumas classes na linguagem Java que podem ser incorporadas a outros projeto para a leitura de um programa nessa linguagem. Das classes geradas, utilizaremos apenas duas: uma com o sufixo *Lexer*, responsável somente pela transformação de texto em uma sequência de *tokens*, e uma com o sufixo *Parser*, que utiliza os *tokens* gerados pela classe anterior em uma hierarquia de classes definida a partir da gramática. Estas classes são usadas no

servidor web descrito na próxima seção.

Entretanto, não é suficiente que o *parser* definido para a DSL reconheça um programa de entrada como válido para a linguagem, uma vez que nem todos os aspectos sintáticos da linguagem podem ser capturados pela gramática. Para evitar a distribuição de novas máquinas de estados que possam deixar a LPSD em um estado inconsistente (Requisito *R.4*), é necessário realizar avaliações adicionais sobre máquina de estados definida. Estas avaliações podem ser inferidas da leitura das seções anteriores:

- Ter um número mínimo de dois estados;
- Não existirem dois estados com o mesmo nome;
- Não existirem duas `RangeDeclarations` referenciando a mesma *feature*;
- Os objetivos de qualidade possuem limites definidos entre zero e dez;
- Se as `RangeDeclarations` referenciam somente *features* numéricas;
- Se o tipo do valor em `FusionRules` é compatível com o tipo da *feature*.

Features de tipo numérico podem ser comparadas com dois tipos diferentes de valores em `FusionRules`: faixas de valores, do tipo `BigRange` e estado, usando o nome de um estado definido. Entretanto isso só é possível se a *feature* numérica possuir uma definição de faixa (`RangeDeclaration`).

Além destas avaliações, devemos garantir ainda que cada estado na definição da máquina de estados permita selecionar ao menos uma configuração válida para a LPSD, caso contrário a especificação não será considerada válida (Requisito *R.4*). No aplicativo Monitor original, os parâmetros de qualidade das configurações eram computados durante o processo de reconfiguração da LPSD [Fernandes, 2012]. Considerando que a LPSD é fechada com relação a suas *features*, decidimos criar uma tabela com os valores de objetivos de qualidade de cada configuração válida previamente computados. Para esta implementação, entretanto, utilizamos somente os valores do parâmetro de confiabilidade da configuração; os valores obtidos foram analisados e transformados de modo a serem compreendidos na faixa entre zero e dez equivalentes aos valores válidos para os objetivo de qualidade dos estados. A partir da criação desta tabela, a avaliação deve apenas testar se existe ao menos uma configuração válida para cada faixa de objetivos de qualidade dentre as configurações existentes na tabela.

Uma vez que a tabela de confiabilidade das configurações válidas possui todas as configurações possíveis para a LPSD, optamos por não apenas testar a existência de ao menos uma configuração válida, mas também identificar todas as configurações válidas

para cada estado e acrescentar esta informação à árvore sintática produzida pelo *parser*. A adição desta informação deve otimizar o processo de seleção de uma nova configuração pela LPSD, removendo a etapa de cálculo de atributos em tempo de execução (Requisito *R.3*), além de permitir a substituição da tabela sem necessidade de atualização para o aplicativo Android (Requisitos *R.1* e *R.4*).

4.2.3 Definindo o Comportamento da DSL

Apesar de a sintaxe da DSL ser composta por elementos de um domínio restrito, o seu comportamento pode parecer falsamente intuitivo para implementadores, que podem acreditar erroneamente que conhecem o domínio da aplicação, e usuários. Em ambos os casos, o entendimento incorreto da DSL é a maior fonte de erros em DSLs [Strembeck e Zdun, 2009]. Mesmo em casos onde pareça desnecessário, definir explicitamente como elementos da DSL interagem para produzir o comportamento definido pelos projetistas pode ser uma ferramenta útil.

Com a adoção desta proposta para a evolução de uma LPSD, é importante garantir que o comportamento prévio [Fernandes, 2012] da aplicação seja mantido, ou seja, que seja possível definir um programa que reproduza o mesmo objetivo original da LPSD. A definição deste programa também pode ajudar a elaborar a documentação do comportamento esperado da DSL.

Esta definição do comportamento da DSL pode ser feita através de descrição textual simples porém precisa e executável, de modelos de fluxo de controle de alto nível, ou de modelos de comportamento detalhados usados para geração do código correspondente. Este processo deve ser realizado até que a definição esteja completa e que não sejam detectados erros ou inconsistências [Strembeck e Zdun, 2009]. Em essência, o objetivo deste passo é definir o algoritmo a ser usado para a interpretação da DSL.

Para a DSL da máquina de estados do sistema Monitor, o comportamento esperado pode ser descrito através de três regras simples, as quais optamos por descrever de forma textual (vide Figura 4.6). Todos os elementos são avaliados na mesma ordem em que são declarados na forma da DSL, de cima para baixo, da esquerda para a direita.

1. Para cada `RangeValue` de cada `RangeDeclaration` é comparado o valor indicado na `RangeValue` com o valor da *feature* indicada na `RangeDeclaration` para verificar se o valor indicado é menor que o valor da *feature*;
   - Em caso afirmativo, o estado associado à `RangeValue` é associado à *feature* indicada na `RangeDeclaration` e a avaliação segue para a próxima `RangeDeclaration`
   - Caso contrário a `RangeValue` seguinte é avaliada;

2. Deve-se registrar um estado intermediário na avaliação da máquina de estados. Quando o estado de uma *feature* for alterado, deve ser realizada uma comparação entre os objetivos de qualidade do estado intermediário e do estado recém-atribuído à *feature*;

    - Caso o objetivo de qualidade do novo estado seja superior ao do estado intermediário, o novo estado substitui o estado intermediário no registrador;
    - Na nossa definição, um objetivo de qualidade de um estado A é maior que o de outro estado B se o limite superior de A é maior que o de B ou, apenas caso os dois sejam iguais, se o limite inferior de A é maior que o de B;
    - Nada é feito caso o objetivo de qualidade do novo estado seja igual ou inferior ao do estado intermediário;

3. Para cada `FusionRule` de cada `StateDeclaration` são avaliadas as `Rules` por comparação entre o valor ou estado indicado para a *feature* também indicada na `Rule`

    - Caso todas as comparações de `Rules` em uma `FusionRule` sejam avaliadas como verdadeiras, o estado intermediário passa a ser o estado que declara a `FusionRule`
    - Caso contrário, a próxima `FusionRule` deve ser avaliada.

Essa sequência deve ser seguida para evitar que a avaliação de uma *feature* mascare o resultado de outra que represente um risco de saúde maior para o indivíduo monitorado sem que esta seja uma exceção. Os objetivos de qualidade e demais propriedades dos estados são utilizados diretamente pelo aplicativo Monitor e serão usados para exibir informações sobre o estado e realizar seleção de uma configuração válida para o estado.

4.3 Análise de Objetivos de Qualidade

Como vimos na seção anterior, cada estado definido na especificação de uma máquina de estados possui associada a si um objetivo de qualidade. Este objetivo é representado por um intervalo contínuo cujos limites inferior e superior não podem passar de zero e dez respectivamente (como observamos na definição da sintaxe concreta na seção anterior, estes intervalos podem ser definidos omitindo um dos dois limites, mas esta notação não implica em extensão desses limites). O intervalo do objetivo de qualidade de um estado é usado para determinar as configurações válidas que a LPSD pode assumir para aquele determinado estado.

Para que esta seleção de configurações possa ser realizada, é necessário determinar o objetivo de qualidade de cada configuração da LPSD. Em Fernandes [2012], o objetivo de qualidade de cada configuração é computado no momento em que uma nova configuração é necessária; esse computo pode ser realizado com o uso de duas abordagens diferentes acrescidas de pesos previamente computados. Como podemos observar na Figura 4.2, a quantidade de configurações possíveis no aplicativo Monitor é relativamente pequena e gerenciável, o que nos permite realizar esta computação em tempo de execução em um dispositivo móvel.

Como forma de otimizar o desempenho da LPSD (Requisito *R.3*), verificamos ser possível computar previamente os objetivos de qualidade das configurações da linha de produtos e armazenar estas informações em uma tabela, permitindo rapidamente a seleção de configurações para a linha de produtos. O sistema então passa a utilizar somente as configurações que existam na tabela, confiando que ela esteja correta. Assim, configurações inválidas podem surgir apenas a partir de erro humano ou ataques.

Para a implementação do sistema Monitor, optamos por substituir as abordagens descritas por Fernandes [2012] pelo uso da informação de confiabilidade (*reliability*) das configurações. O cálculo de confiabilidade das configurações da LPSD é descrito por Nunes [2012] e foge ao escopo do trabalho corrente. Observamos, entretanto, que todos os valores de confiabilidade obtidos estavam compreendidos entre $0,94$ e $0,99$. Para simplificar a seleção de configurações pelo especialista no domínio, optamos por trabalhar com uma faixa de objetivos de qualidade variando entre 0 e 10. Logo, as confiabilidades das configurações foram transformadas em valores dentro desta escala utilizando a seguinte fórmula $O = (C - 0.94) * 200$, onde O é o objetivo de qualidade e C é a confiabilidade da configuração.

As abordagens anteriores podem ser suportadas através da adição de novas propriedades na definição de um estado para indicar a abordagem a ser utilizada bem como os pesos a serem adotados. Um esboço desse suporte foi realizado ao longo deste trabalho, mas foi posteriormente destacado em favor do uso da confiabilidade.

4.4 Arquitetura

A arquitetura de um sistema computacional pode ser descrita como o particionamento prudente do sistema em partes com os respectivos relacionamentos entre essas partes [Garlan et al., 2010].

Considerando que a DSL é apenas uma ferramenta auxiliar para permitir a flexibilização dos objetivos da LPSD, buscamos definir uma arquitetura complementar cujos componentes acessórios à DSL pudessem ser desenvolvidos de forma independente. A

arquitetura proposta apresenta um servidor que contém funções a serem acessadas tanto pela LPSD quanto pelo especialista no domínio (através de um aplicativo cliente). Ao especialista no domínio, a interface do cliente deve permitir a recuperação e alteração dos objetivos da LPSD; caso seja possível personalizar uma instância da LPSD (Requisito *R.8*), esta interface também deve prover os meios necessários para a seleção do objetivo individual a ser trabalhado. Para a LPSD, o servidor provê os meios para notificar a LPSD sobre atualizações nos seus objetivos e para permitir a recuperação os objetivos atualizados, conferindo independência de presença do dispositivo que contém a LPSD no momento da atualização (Requisito *R.2*).

A Figura 4.8 apresenta esta arquitetura complementar à LPSD.

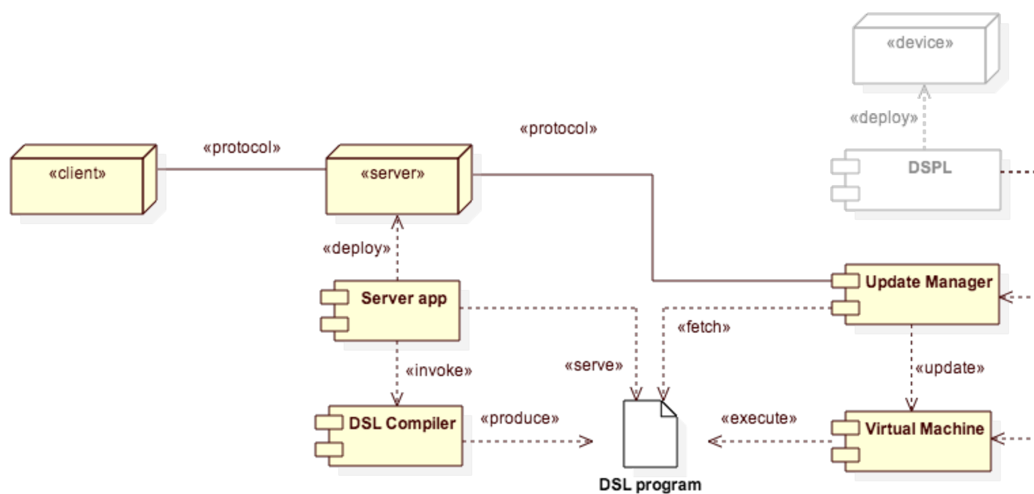

Figura 4.8: Diagrama de distribuição proposto para flexibilização de LPSDs

Nesta figura, podemos observar a vinculação de dois novos componentes à LPSD propriamente dita (componentes anteriores estão representados por um único componente - *DSPL*). O gerente de atualização é responsável por atualizar os objetivos definidos pelo especialista no domínio usando a DSL quando notificado de atualizações pelo servidor através de dois passos: (1) obter uma cópia dos novos objetivos junto ao servidor, e (2) disponibilizar à máquina virtual os novos objetivos. A máquina virtual é responsável por interpretar os objetivos sempre que necessário para o funcionamento da LPSD.

No lado do servidor, podemos observar a presença de um compilador para a DSL; este será responsável por traduzir as informações representadas através da DSL em instruções para a máquina virtual, com o objetivo de reduzir o conjunto de instruções e operações que devem ser realizados pela LPSD (Requisito *R.3*). Van Deursen et al. [2000] também descreve que a compilação de DSLs permite realizar detecção de erros, análise estática e otimizações no nível do domínio.

Para o sistema Monitor, optamos pela implementação de uma aplicação web para a plataforma Java, dado o uso da ferramenta ANTLR já mencionada que produz *parsers* na linguagem utilizada por esta plataforma. Esta opção também simplifica o desenvolvimento da solução, pois o cliente pode ser qualquer navegador web padrão.

Como forma de otimizar a execução e validação dos objetivos da LPSD definidos através da DSL (Requisitos *R.3*, *R.4* e *R.7*), sugerimos transformar estas especificações em instruções imperativas que representam o comportamento esperado da DSL, descrito na seção anterior. Este procedimento se assemelha com o subpadrão de transformação de código descrito por Mernik et al. [2005] exceto que a linguagem alvo da transformação não é gráfica ou textual, mas uma sequência de bytes que pode facilmente ser interpretada por outro componente de código, como uma máqina virtual.

4.4.1 Definição da Máquina Virtual

Uma máquina virtual cria uma camada de abstração entre o que é desejado permitir evolução na LPSD e a definição e implementação da DSL propriamente dita, permitindo assim que a própria DSL possa ser evoluída ou mesmo substituída sem afetar a implementação da LPSD e vice-versa. É importante lembrarmos que a DSL definida na seção anterior foi avaliada de forma preliminar por um especialista no domínio médico e que, em trabalhos futuros, uma nova linguagem pode ser definida em substituição à desenvolvida neste trabalho, e mesmo que o próprio conhecimento médico pode evoluir a ponto de requerer mudanças a nível da DSL.

Optamos por introduzir uma pequena máquina virtual no sistema para o processamento da máquina de estados devido a algumas limitações encotradas. A implementação do Android, por exemplo, não permite a substituição de código Java que já esteja carregado em memória, sendo necessário esforço considerável na implementação de um *ClassLoader* ou a interrupção do aplicativo móvel para realizar a atualização da máquina de estados. Além disso, apesar do *bytecode* Java ser portável, não existe suporte direto para sua execução em outras plataformas móveis, exigindo a implementação de uma JVM completa nestes casos. A introdução de uma VM pequena e gerenciável nos permite contornar estes problemas.

No cenário de evolução e flexibilização de objetivos de uma LPSD, a adoção de uma máquina virtual tem também o objetivo de otimizar o desempenho do processamento das instruções associadas ao objetivo, minimizando o esforço realizado pela LPSD para interpretação destes. Isto é alcançado transformando a descrição do comportamento da DSL, discutida na seção anterior, em sequências de instruções que representam as operações descritas para o comportamento definido. Em trabalhos futuros, a adoção da máquina

virtual também poderá ajudar no suporte a objetivos ou comportamentos que não foram previstos inicialmente para a DSL ou para a LPSD.

As instruções para esta máquina virtual variam de acordo com as necessidades da LPSD e da DSL, e o processo de identificação das instruções é semelhante ao descrito no início da seção anterior, podendo o número de instruções crescer de acordo com mudanças nos requisitos. Elaboramos uma DSL composta por instruções capazes de representar todas as operações descritas para o comportamento da DSL da máquina de estados e, a partir do trabalho de Craig [2006], elaboramos um conjunto de regras a serem observadas na definição das instruções da máquina virtual:

1. Instruções podem definir operações simples ou primitivas (como operações aritméticas e desvios), ou complexas ou de alto nível (como criação e manipulação de objetos e vetores);

2. Quando uma sequência de instruções primitivas aparece repetidamente no código, pode ser útil combiná-las em uma única instrução de alto nível;

3. Por outro lado, instruções com muitos argumentos podem ser muito específicas e de pouca utilidade, sendo conveniente separar as operações destas instruções em outras mais simples e reusáveis;

4. Manipulação de estruturas de dados complexas (tais como imagens ou matrizes), acesso a bancos de dados e chamadas a procedimentos remotos podem requerir a definição de instruções de alto nível.

Comumente, uma máquina virtual é implementada através de um laço principal, conforme visto na Seção 2.4. Ao ser implementado em uma linguagem orientada a objetos, podemos refatorar este laço usando o padrão *Strategy* [Gamma et al., 1994], conforme ilustrado na Figura 4.9 que apresenta o mesmo laço da Listagem 2.1 estruturada segundo este padrão. Nesta versão, cada subclasse da classe `Instruction` é responsável pelo código a ser executado para cada instrução. Este modelo é vantajoso pois elimina o risco de uma implementação incorreta da VM posicionar o ponteiro de instrução em uma posição inválida (em um vetor de *bytes* ou arquivo, por exemplo), uma vez que o ponteiro de instrução passa a apontar diretamente para um objeto que representa uma instrução estruturada válida.

A partir da definição do comportamento da DSL, definimos as instruções da máquina virtual. As instruções, definidas abaixo, A Figura 4.10 apresenta o modelo de classes das instruções da máquina virtual. As seguintes instruções foram definidas para nossa VM:

- *CompareInstruction*, `COMPARE`, compara o valor de uma *feature* de informação a um valor armazenado no objeto que representa a instrução;

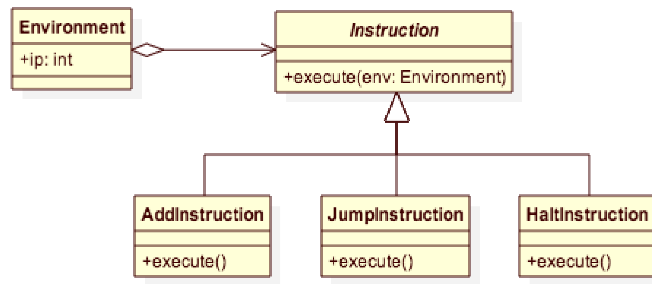

Figura 4.9: Modelo de classes para realização do laço principal de uma VM

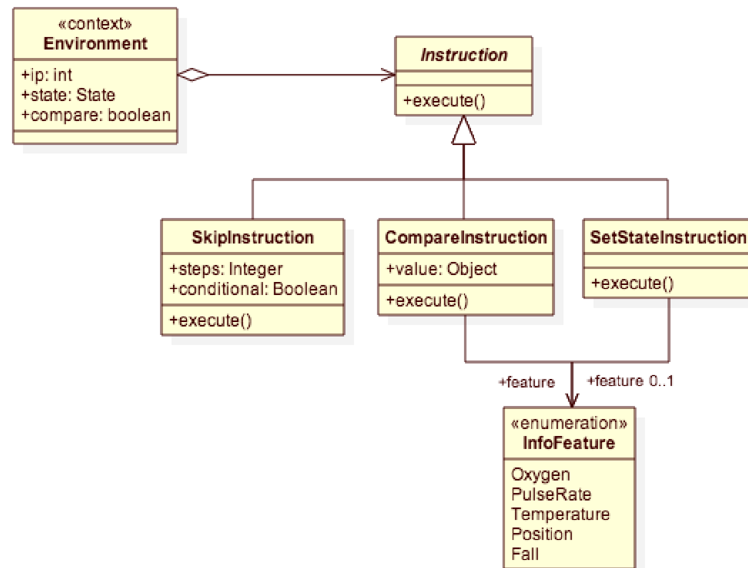

Figura 4.10: Modelo de classes das instruções da máquina virtual

- *SkipInstruction*, `SKIP`, instrui a máquina virtual a saltar um determinado número de instruções, salto este que pode ser condicionado ou não ao resultado da última instrução de comparação;

- *SetStateInstruction*, `SETSTATE`, é uma instrução de alto nível que atualiza o estado intermediário resultante da execução da máquina de estados, cuja regra varia dependendo da indicação ou não de uma *feature* nos argumentos da instrução:

  - Se a instrução não possuir uma *feature* associada, o estado intermediário é substituído pelo estado indicado na instrução;

  - Caso contrário, o estado é atribuído como estado da *feature* indicada e é realizado um novo teste para determinar se o objetivo de qualidade do novo estado é maior que o do estado intermediário. Neste caso, o estado intermediário é substituído somente se o objetivo de qualidade do novo estado for maior (vide Seção 4.2.3).

Aqui, optamos por não simplificar a instrução de comparação (`COMPARE`) que, como será visto mais adiante, é sempre seguida por uma instrução de salto (`SKIP`) condicionado ao resultado da comparação. Esta decisão é arbitrária e não existem impedimentos para sua revisão. O modelo orientado a objetos também nos permite adicionar novas instruções no futuro, caso seja necessário, através da implementação de uma nova classe representando o comportamento da instrução.

Além do ponteiro de instrução, a partir da observação do comportamento da VM, podemos observar a necessidade de outros quatro registradores durante a execução da VM:

- um registrador lógico para indicar o resultado da última instrução de comparação;
- um registrador do estado intermediário da execução da máquina de estados;
- um conjunto de registradores de estados, um para cada *feature* de informação implementada na LPSD;
- um conjunto de registradores de valores, um para cada *feature* de informação implementada na LPSD, como forma de congelar os valores das *features* durante a execução da máquina de estados.

Não há necessidade de uma instrução de saída para esta máquina virtual pois a instrução `SKIP` é usada somente para avançar o ponteiro de instrução, ou seja, sem criar laços de repetição. A máquina virtual deverá assumir que a execução da máquina de estados chegou ao fim quando o ponteiro de instrução possuir um valor maior que a quantidade de instruções disponíveis para executar. Neste momento, o valor do registrador de estado intermediário deve ser retornado como o estado de saúde identificado para o indivíduo.

Para flexibilizar os objetivos da LPSD do sistema Monitor, é suficiente que a execução da máquina virtual retorne apenas o estado de saúde em que se encontra o indivíduo monitorado a partir a comparação dos valores das *features* de informação utilizando-se das instruções enviadas ao aplicativo Monitor através do servidor de aplicação. A partir do estado retornado, a LPSD poderá escolher uma das configurações possíveis associadas ao estado (determinada pelo compilador, como veremos mais adiante) e realizar a operação de reconfiguração da LPSD.

4.4.2 O Processo de Compilação da DSL

O propósito do compilador da DSL proposta é realizar a transformação da representação produzida com o uso da DSL em instruções para a máquina virtual. Para o sistema Monitor, todas as instruções referentes à máquina de estados são geradas a partir

das declarações de faixa (`RangeDeclaration`) e das regras de fusão (`FusionRule`), seguindo as regras descritas na Seção 4.2.3.

Assim, determinamos que cada `RangeDeclaration` será transformada em precisamente 4 instruções para cada estado referenciado na declaração (cada `RangeValue` que aponta um estado), exceto pela última referência que terá somente 3 instruções; a Listagem 4.4 apresenta uma abstração da sequência de instruções geradas para cada `RangeValue`. Nesta listagem, os valores entre chaves angulares representam variáveis a serem resolvidas durante o processo de compilação da DSL. Sendo E o número de estados referenciados, computamos a quantidade total de instruções N a serem geradas para uma `RangeDeclaration` como sendo $N = 4 * E - 1$.

```
1 COMPARE <feature> <faixa de valores>
2 SKIP 2 IF FALSE
3 SETSTATE <estado> <feature>
4 SKIP <N -= 4>
```

Listagem 4.4: Abstração do conjunto de instruções gerada para uma RangeDeclaration

Observe que o valor de N deve ser reduzido em 4 antes de ser atribuído à instrução `SKIP` na linha 4. Assim, a última instrução gerada para uma `RangeDeclaration` indicará um salto negativo (-1). Esta instrução deve ser removida, dado que não queremos saltar para trás e sim seguir para a próxima instrução. Também se faz necessário atualizar a última instrução do tipo da linha 2 que foi gerada para reduzir seu salto de 2 para 1, para compensar a exclusão da última instrução `SKIP`. A ordem de declaração das faixas no código não deve afetar o resultado final desta execução.

Por sua vez, a ordem de declaração das `FusionRules` afeta o resultado; suas instruções são produzidas na mesma ordem em que se apresentam na especificação da máquina de estados (ou seja, a ordem em que os estados são declarados também afeta o resultado). Estas podem ser consideradas não apenas regras complementares mas também regras de exceção, e podem inclusive alterar o estado resultante para um de menor objetivo de qualidade. De forma semelhante às `RangeDeclarations`, serão geradas 2 instruções para cada `Rule` que compõe a `FusionRule` acrescida de uma instrução adicional ao final. Sendo R a quantidade de `Rules` em uma `FusionRule`, definimos então a quantidade total de instruções para uma `FusionRule` N como $N = 2 * R + 1$. A Listagem 4.5 apresenta as instruções que são geradas para cada `Rule` de uma `FusionRule` (linhas 1 e 2) e ao final do processamento da própria `FusionRule`. As reticências nesta notação indicam apenas que as instruções nas linhas anteriores se repetem conforme o necessário para o processamento da `FusionRule`, enquanto os valores entre chaves angulares são variáveis resolvidas pelo compilador durante a compilação da DSL.

```
COMPARE <feature> <valor>
SKIP <N -= 2> IF FALSE
...
SETSTATE <estado>
```

Listagem 4.5: Série de instruções gerada para uma regra de fusão

A última instrução nesta listagem (linha 4) é gerada após a repetição das demais instruções para cada `Rule` causando a atualização do estado resultante caso todas as comparações da regra sejam verdadeiras. A última regra de fusão avaliada irá produzir saltos que farão o ponteiro de instrução assumir um valor superior à quantidade de instruções definidas para o programa. Como vimos anteriormente, nestas condições a máquina virtual assumirá que a execução da máquina de estados foi encerrada e informará o estado obtido à LPSD.

Além das instruções, como forma de otimizar o desempenho e garantir o *safety* da LPSD (Requisitos *R.3* e *R.7*), transferimos para este compilador a tarefa de determinar o conjunto de configurações da LPSD que são válidas para atingir o objetivo de qualidade de cada estado. Assim, diminuímos o processamento necessário no aplicativo Monitor para a seleção de um estado válido computando todas as configurações possíveis para um estado a partir do seu objetivo de qualidade utilizando uma ou mais abordagens para calculo do objetivo de qualidade de uma configuração. Isso também acrescenta uma flexibilidade adicional à LPSD pois permite a alteração das regras para seleção das configurações que satisfazem os objetivos de qualidade definidos apenas no compilador, sem afetar a implementação do aplicativo.

Como vimos anteriormente, caso o compilador não consiga selecionar ao menos uma configuração válida para qualquer um dos estados definidos na especificação da máquina de estados, esta será considerada inválida e não produzirá uma atualização, retornando uma mensagem para o especialista no domínio indicando o motivo da falha no processo de atualização.

4.5 Implementação

Com a produção dos artefatos do projeto do domínio, procedemos à implementação da LPSD. Nesta etapa, realizamos a implementação de todos os componentes propostos na arquitetura resultante do projeto do domínio.

Nesta seção, discutiremos alguns detalhes sobre a implementação da LPSD que não foram abordados em seções anteriores.

4.5.1 Aplicativo Móvel

Conforme mencionamos anteriormente, o aplicativo móvel do sistema Monitor, utilizado na implementação da técnica apresentada no capítulo anterior, foi desenvolvido por Fernandes [2012]. O aplicativo foi estruturado como apresentado na Figura 4.3 enquanto a Figura 4.11 apresenta a mesma representação da arquitetura com as alterações propostas neste trabalho.

Figura 4.11: Arquitetura completa do sistema Monitor

Em relação ao trabalho inicial de Fernandes [2012], o aplicativo Android foi atualizado para se beneficiar de uma API atualizada (passando a usar o SDK 16 em detrimento do 7) e utilizar uma interface gráfica mais enxuta e aderente aos padrões de interface sugeridos pelo Google[2]. A Figura 4.12 apresenta a tela inicial reestruturada para o aplicativo móvel.

Em relação à arquitetura discutida na seção anterior, o aplicativo Android se comunica com o servidor de aplicações através de uma interface web/HTTP para realizar duas operações: uma para registro do dispositivo, necessário à etapa de notificação de atualizações, e outra para download do arquivo contendo a nova definição da máquina de estados. O mesmo ID de paciente é usado pelo dispositivo para permitir a distinção entre os diferentes perfis do indivíduo monitorado e atualização correta da máquina de estados (Requisito *R.8*).

O arquivo contendo a especificação da máquina de estados não é imediatamente aplicado pelo aplicativo. Previamente é realizada uma verificação de integridade do arquivo através da sua leitura. Isso é possível pois cada tipo de dado armazenado no arquivo é precedido por um *byte* que confirma o tipo do dado sendo lido. Por exemplo, um valor do tipo inteiro de 4 *bytes* é armazenado no arquivo usando 5 *bytes*; o primeiro destes *byte*

[2] http://developer.android.com/design/patterns/new.html

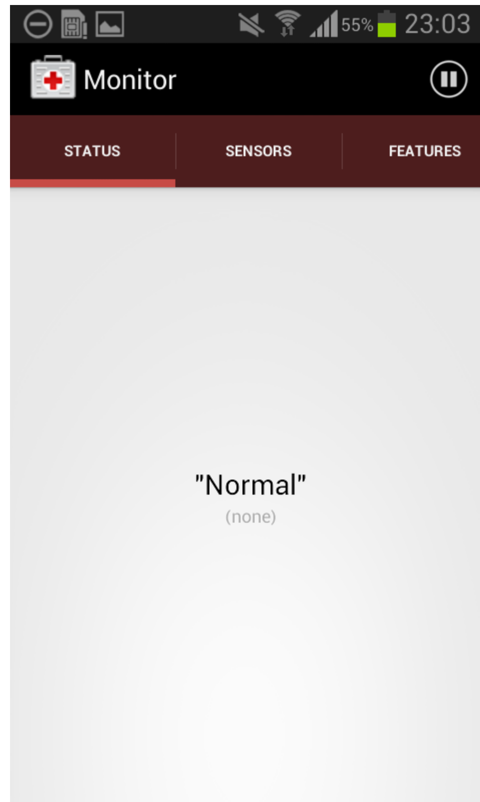

Figura 4.12: Tela inicial atualizada do aplicativo Monitor

é uma constante que representa o tipo de dado (por exemplo, a constante 4 indica um valor do tipo inteiro com 4 *bytes*) e é usado para verificar o tipo correto do dado sendo lido. A leitura do arquivo deve seguir a mesma ordem da sua geração. A interpretação correta do arquivo irá reproduzir um vetor de instruções com seus respectivos parâmetres que pode ser usado pela máquina virtual. O arquivo também é armazenado em uma área protegida do dispositivo para rápida inicialização, caso o aplicativo seja encerrado.

4.5.2 Servidor de Aplicação WEB

Optamos pelo desenvolvimento de uma aplicação WEB para construir as interfaces de comunicação entre o especialista no domínio e o aplicativo móvel, em atendimento à necessidade de atualização remota dos objetivos de qualidade (Requisito *R.2*) e à restrição de que somente um especialista no domínio poderia realizar alterações (Requisito *R.6*). Para comportar os *parsers* gerados pelo ANTLR, utilizamos um servidor web para plataforma Java (Apache Tomcat 7.0[3]).

O projeto da aplicação WEB foi iniciado com as classes geradas para a leitura e compilação de espeficiações da máquina de estados. O código responsável pela leitura

[3]http://tomcat.apache.org/

é gerado pela ferramenta ANTLR a partir da definição dada na Listagem 4.3 enquanto o compilador é produzido com as especificações da Seção 4.4.2. Neste projeto, optamos por desenvolver o compilador em conjunto com a aplicação web porém com separação suficiente que permitisse reaproveitar o compilador em outro contexto. Outra opção teria sido desenvolver a funcionalidade de compilação à parte e posteriormente integrar esta à aplicação WEB.

A Figura 4.13 apresenta a interface de atualização dos objetivos da LPSD, a ser utilizada pelo especialista no domínio. Para esta versão da interface não nos preocupamos com controle de acesso do usuário, necessário para evitar alterações indesejadas nas máquinas de estado (Requisito *R.6*).

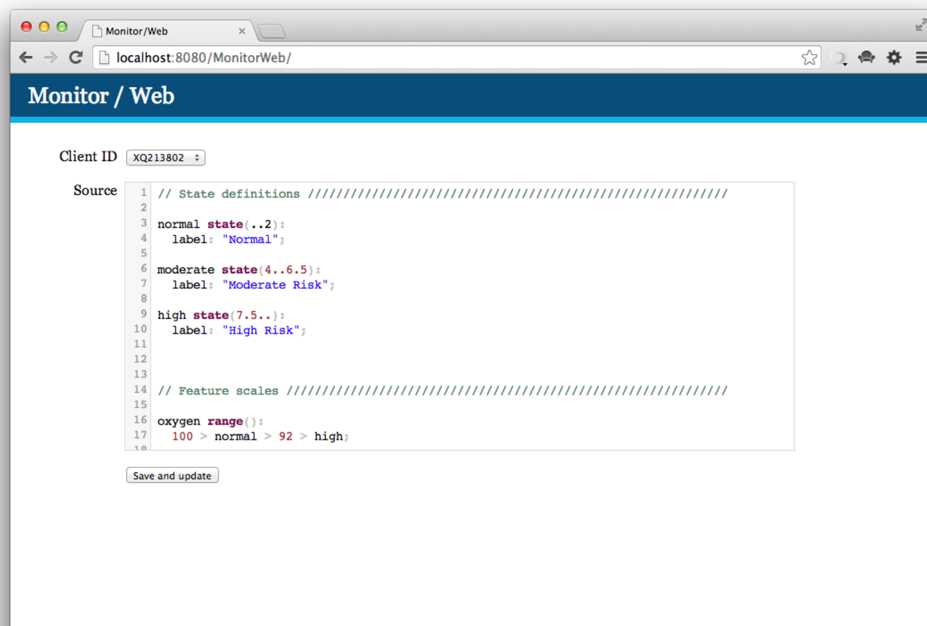

Figura 4.13: Tela de entrada de dados do especialista no domínio

Esta interface web possui apenas um campo para entrada do código referente à especificação da máquina de estados e uma caixa de seleção para realizar a diferenciação entre os indivíduos monitorados através de um ID do paciente(Requisito *R.8*). A seleção de um ID diferente carrega o último código validado para o respectivo ID.

Notificação de Atualizações

Uma vez definido um novo programa para a máquina de estados de um indivíduo, é necessário refletir esta atualização no dispositivo móvel onde se encontra a LPSD. Para isso, não é viável que o dispositivo realize polling do servidor para determinar se existem

atualizações para o seu programa; isto causaria um gasto excessivo de recursos como a bateria, o que resultaria em uma redução da autonomia do dispositivo (Requisito *R.3*).

Para evitar este gasto, utilizamos um recurso disponibilizado pelo Google para a plataforma Android, chamado Google Cloud Messaging (GCM), para realizar a notificação de atualizações de especificações da máquina de estados. A Figura 4.14 apresenta um diagrama de sequência com os passos necessários para o registro do aplicativo móvel com o serviço de notificação. No caso do GCM, antes de executar estes passos, é necessário que o desenvolvedor tenha uma conta de usuário do Google e realize o cadastro da aplicação que irá enviar/receber as mensagens. Este cadastro irá produzir uma chave alfanumérica (GCM_KEY) que deve ser conhecida do aplicativo Android. Este cadastro deve ser realizado apenas uma vez.

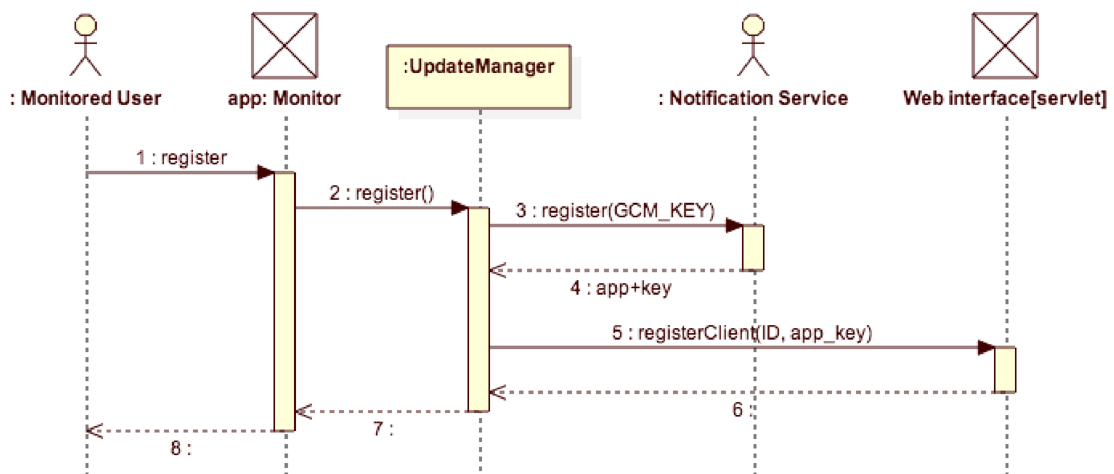

Figura 4.14: Processo de registro da máquina de estados para receber atualização

O processo de registro para notificação é iniciado pelo aplicativo no dispositivo móvel. O aplicativo deve entrar em contato com o servidor do GCM através de uma requisição HTTP informando a chave alfanumérica do programa para o qual deseja se registrar para o recebimento de notificações. Esta chave alfanumérica é registrada previamente pelo desenvolvedor junto ao Google e inclusa no código do aplicativo móvel. A resposta do servidor do GCM irá conter uma nova chave alfanumérica, esta identificando o aplicativo instalado naquele dispositivo em particular. Esta chave deve ser enviada para o servidor de aplicação, também através de uma chamada HTTP, junto com o ID do paciente, que identifica o indivíduo monitorado no servidor de aplicações, para completar o processo de registro e permitir que notificações sejam recebidas pelo dispositivo. O servidor de aplicação irá armazenar a chave enviada pelo servidor do GCM associada ao ID do paciente.

O servidor de notificações do GCM não é requisito essencial para o desenvolvimento da solução. Na sua ausência, outros mecanismos poderiam ser usados para implementar o padrão *Observer*, também conhecido como *Publisher-Subscriber* [Gamma et al., 1994], em um dispositivo móvel.

Processo de Atualização

Uma vez que o dispositivo móvel esteja registrado junto ao servidor de aplicação, a chave alfanumérica de identificação da aplicação no dispositivo, enviada no passo 5 do diagrama de sequência da Figura 4.14, é usada para notificar o aplicativo móvel da existência de uma máquina de estados atualizada para este paciente.

A atualização dos objetivos da LPSD do sistema Monitor procede através do processo apresentado na Figura 4.15. Nesta figura, o ator "Notification Service" separa os componentes que estão no lado do servidor (à esquerda da imagem) dos que estão no lado do dispositivo móvel (à direita da imagem).

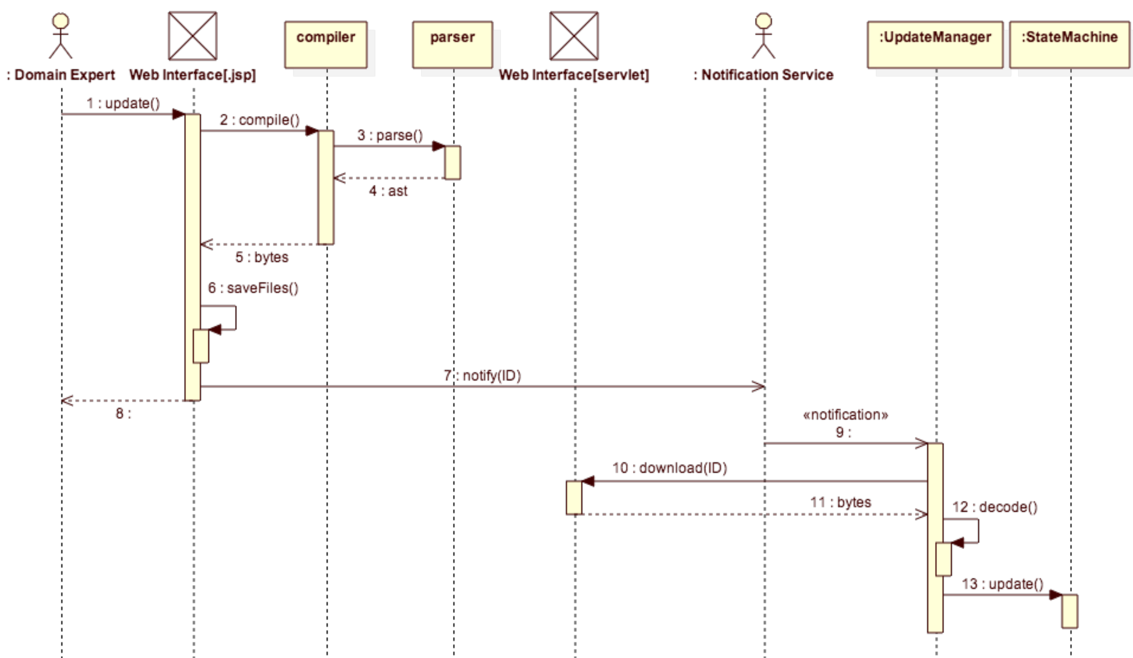

Figura 4.15: Processo de atualização da máquina de estados

O processo de atualização é sempre iniciado pelo especialista no domínio, atráves da interface apresentada anteriormente na Figura 4.13, ao pressionar o botão de atualização. O código-fonte informado pelo especialista é processado para verificar erros sintáticos e semânticos conforme descrito no capítulo anterior e o especialista é notificado no caso de erros; em caso de sucesso, o resultado da compilação convertido em uma sequência representativa de bytes armazenada associada ao ID do paciente monitorado.

A notificação de nova atualização depende do registro do dispositivo descrito na seção anterior. De posse do identificador do dispositivo junto ao GCM para o paciente identificado, é possível realizar uma chamada HTTP ao servidor do GCM criando uma notificação para o aplicativo. Esta notificação será entregue independente do tempo decorrido, caso o aparelho esteja desligado ou desconectado de uma rede com acesso à internet (Requisito *R.5*). Esta notificação também não será recebida em duplicidade, caso o especialista no domínio altere uma segunda vez a máquina de estados antes da primeira atualização ter sido efetivada. Ao receber esta notificação, o aplicativo entra em contato com o servidor de aplicação e transfere as regras atualizadas. Estas ainda terão sua integridade verificada antes de serem disponibilizadas à máquina virtual, conforme discutido na seção aterior.

4.6 Resumo

Ao longo deste capítulo, descrevemos um processo para permitir a flexibilização dos objetivos de uma linha de produtos de software dinâmica utilizando uma linguagem específica de domínio. Pudemos verificar que a análise de domínio realizada para a construção da LPSD pode ser reusada para determinar as abstrações que serão usadas pela DSL e que a introdução desta na LPSD cria um novo ponto de variação aberto, ou seja, para o qual não conhecemos todas as variantes possíveis. Em seguida, acompanhamos o processo de definição de uma DSL, de uma máquina virtual, e de um compilador que permite a tradução de especificações realizadas usando a DSL para programas na máquina virtual definida. Uma arquitetura complementar foi apresentada para suportar as necessidades de atualização e flexibilização da LPSD.

Acompanhamos também, ao longo de cada seção deste capítulo, a resolução dos passos deste processo aplicados a uma LPSD do domínio médico para monitoração de indivíduos utilizando uma RSCH. No próximo capítulo, discursaremos sobre a experiência de implementação desta especificação para permitir a flexibilização dos objetivos da LPSD do sistema Monitor.

Capítulo 5

Avaliação

A validação deste trabalho é feita através de uma simulação, que tem como objetivo analisar o comportamento da aplicação. Isso se faz necessário para que possamos verificar se o comportamento anterior da aplicação é mantido [Fernandes, 2012], bem como verificar o comportamento adicionado como parte deste trabalho. Adicionalmente, desejamos verificar o desempenho geral da aplicação, em especial o impacto da VM da máquina de estados sobre a reconfiguração da linha de produtos dinâmica. Esta avaliação também servirá de referência para futuras evoluções do projeto do sistema Monitor como um todo, uma vez que trata-se de um projeto em andamento.

Este capítulo apresenta os detalhes da avaliação realizada sobre o sistema Monitor com as alterações implementadas vistas no Capítulo 4. O restante deste capítulo está organizado da seguinte forma: a Seção 5.1 apresenta o método utilizado para controlar a realização da avaliação, na Seção 5.2 são apresentados os contextos considerados para a avaliação, enquanto os resultados da avaliação são discutidos na Seção 5.3. Por fim, as possíveis ameaças à validade desta avaliação são apresentados e discutidos na Seção 5.4.

5.1 *Goal Question Metric* (GQM)

Por se tratar de um sistema que lida com a vida humana, é necessário garantir certas qualidades do sistema para evitar que ele traga riscos ao indivído sendo monitorado. Podemos identificar, dentre outros, dois aspectos essenciais que podem trazer riscos para uma aplicação desta natureza: eficiência e *safety*. Aqui optamos por fazer uso do termo em inglês para evitar a ambiguidade semântica entre os termos *safety* e *security*, que se traduzem para o português com o mesmo termo.

Devido à sua estruturação sistemática, optamos pelo uso do método *Goal Question Metric* (GQM) [Basili et al., 1994] para orientar a condução desta avaliação. Este método nos permite estruturar a avaliação de forma *top-down*, a partir da identificação dos ob-

jetivos da avaliação e então os dados que poderão ser usados para definir esses objetivos operacionalmente. Além disso, este método ainda provê um *framework* para interpretação dos dados em relação aos objetivos definidos.

No passo inicial do GQM, definimos a Tabela 5.1 que resume o objetivo de qualidade da avaliação dividido em seus aspectos.

Purpose	Avaliar
Issue	a eficiência e o *safety* do
Object	sistema Monitor implementado com MEIOq
Viewpoint	Engenheiro de aplicação
Context	Dados simulados

Tabela 5.1: Definição dos objetivos de avaliação.

5.1.1 Questões e Métricas

Com o objetivo definido, o passo seguinte é identificar questões que consideramos relevantes para a avaliação do sistema. Levando em consideração a automonia de uma LPSD e, em especial, de um sistema de RSCH, elaboramos três questões que consideramos relevantes esta avaliação. A partir das questões identificadas, foram definidas métricas que serão usadas para explicar como as questões serão avaliadas e respondidas.

- **Q1**. Qual é a capacidade de resposta do sistema Monitor?
  - **M1.1**. Qual o tempo médio para execução da máquina de estados?
  - **M1.2**. Qual o tempo médio que o sistema leva para mudar para uma nova configuração?
  - **M1.3**. Qual o tempo médio para recebimento da notificação de atualização disponível?
  - **M1.4**. Qual o tempo médio que necessário para atualização da máquina de estados?
- **Q2**. Quais os recursos consumidos pelo sistema Monitor?
  - **M2.1**. Qual o tempo de vida médio do sistema sem recarrega da bateria?
  - **M2.2**. Qual a média de consumo de memória do sistema Monitor?
  - **M2.3**. Qual a média de espaço em disco utilizado pela máquina de estados?
- **Q3**. O sistema Monitor é seguro (*safety*)?

– **M3.1**. Existe ao menos uma configuração válida para cada estado definido?

Para avaliar a capacidade de resposta do sistema, optamos por medir o tempo gasto com a reconfiguração da LPSD em duas etapas: o tempo de execução da máquina de estados (*M1.1*), que contém a lógica para determinar o estado de risco de sáude do indivíduo monitorado e pode ser alterada pelo especialista no domínio, e o tempo gasto para a reconfiguração da LPSD (*M1.2*). Devido à capacidade de substituição da máquina de estados, julgamos importante avaliar o tempo necessário para que uma notificação de atualização seja recebida pela LPSD (*M1.3*) e para que a nova máquina de estados comece a ser efetivamente utilizada pelo sistema (*M1.4*). Todos esses tempos serão computados em média devido à imprevisilidade de contextos ao qual o sistema está submetido; para as métricas *M1.1* e *M1.2*, devidos aos dados recebidos dos sensores e diferentes configurações que podem ser sintetizadas, enquanto as métricas *M1.3* e *M1.4* estão sujeitas às condições de rede.

Também consideramos necessário avaliar o consumo de recursos do dispositivo pelo sistema. Por ser executado em um dispositivo móvel, a autonomia de bateria deve ser considerada ao usar o sistema. O consumo de mémoria e o espaço em disco utilizado pela máquina de estados devem ser avaliados devido à possibilidade de substituição da máquina de estados, já mencionada anteriormente.

Finalmente, é importante garantir que o sistema não representará um risco para o indivíduo monitorado. Para isso, precisamos garantir que, para cada objetivo de qualidade definido em um estado na máquina de estados, a LPSD será capaz de identificar ao menos uma configuração válida que atende ao dito objetivo de qualidade.

5.2 Contexto e Cenários

Uma vez concluída a elaboração do GQM, procedemos à definição dos cenários que serão usados para a simulação. De modo a conferir maior autonomia, o sistema foi adaptado para ser capaz de simular o recebimento de dados de sensores lidos de um arquivo. Estes dados são transferidos para os mesmos mecanismos usados internamente para a recepção e tratamento de dados de sensores reais recebidos por Bluetooth.

A leitura de dados biométricos advindo de sensores monitorando um indivíduo é errática e foge ao nosso controle. Tal imprevisibilidade dificulta a avaliação da capacidade de transição de estados em resposta aos dados provenientes desses mesmos sensores, ou seja, ao usar dados reais, podemos não verificar que o sistema é capaz de realizar todas as transições definidas em dada máquina de estados.

Em consideração a isso, foi feita a opção por realizar esta avaliação apenas com a simulação de tais dados biométricos. Os dados usados para composição desse cenário

de avaliação serão construídos com base nos dados reais coletados de um paciente do Beth Israel Hospital - MIT[1], mais especificamente do paciente identificado como MIMIC-055N-ALL-ANN, uma senhora de 63 anos com histórico de hipertensão submetida a uma cirurgia de ponte de safena[2]. Apesar do arquivo citado possuir dados relativos a uma hora de monitoração, foi verificado que nenhum trecho dos dados do paciente possuia alterações substanciais para provocar alguma das transições definidas para a máquina de estados, por este motivo selecionamos um intervalo relativo aos 15 minutos iniciais da medição e o adaptamos de modo que estes pudessem provocar todas as transições previstas para a máquina de estados.

Para realizar a simulação, definimos a máquina de estados na Listagem 5.1. Esta máquina é semelhante à implementada por Fernandes [2012] originalmente para a LPSD, salvo alterações nos objetivos de qualidade dos estados e a inclusão de transições para o estado moderado na avaliação das *features* de *PulseRateInfo* e *OxygenInfo*. É importante observar que deixamos explicitamente de fora desta definição instruções referentes ao uso das *features PositionInfo* e *FallInfo*, ambas advindas do sensor acelerômetro, devido à ausência deste tipo de dado nas fonte de dados do Beth Israel Hospital - MIT.

Tratamos simultaneamente a coleta de dados sobre o consumo de recursos e os tempos de resposta do sistema. Para isso, o programa foi injetado com curtas instruções para registro de tempo e, periodicamente, avaliação da memória e a bateria do dispositivo. Sobre a memória usada pelo aplicativo, optamos por coletar o dado descrito como memória *private dirty*, por se tratar da memória alocada exclusivamente para o processo e que não pode ser paginada [Google, 2014b]. Dados sobre a bateria foram coletados a cada unidade percentual consumida junto com a respectiva hora utilizando um mecanismo fornecido pela própria plataforma Android para notificação de alterações no estado da bateria. Os cenários foram realizados deixando o sistema em execução, desconectado de qualquer adaptador de energia, até a exaustão da bateria.

```
1  // State definitions ///////////////////////////////////////////////////////
2
3  normal state(..1):
4    label: "Normal";
5
6  moderate state(5..6.1):
7    label: "Moderate Risk";
8
9  high state(8..):
10   label: "High Risk";
11
```

[1] http://ecg.mit.edu/

[2] http://physionet.org/physiobank/database/mghdb/patient-guide.shtml#mgh055

```
12
13
14 // Feature scales ///////////////////////////////////////////////////////////
15
16 oxygen range():
17   100 > normal > 93 > moderate > 89 > high;
18
19 pulse range():
20   200 > high > 160 > moderate > 150 > normal > 70 > moderate > 50 > high;
21
22 temperature range():
23   50.0 > high > 38.5 > moderate > 37.5 > normal > 35.0 > moderate > 30.0 >
       high;
```

Listagem 5.1: Código da máquina de estados usada na avaliação

O tamanho da máquina de estados pode ser avaliado em separado, mesmo diretamente no servidor, e sem a necessidade de envio para o dispositivo móvel, dado que não existem diferenças entre o tamanho dos arquivos armazenados em ambos. Na ausência de acompanhamento médico para a definição de outras máquinas de estados, optamos por avaliar somente o tamanho da máquina de estados de teste. Com relação ao tempo para atualização da máquina de estados, optamos por dividir o tempo de atualização coletado em duas partes: uma referente ao tempo necessário para a recepção da mensagem de notificação de atualização (métrica *M1.3*, por esta depender de condições fora do controle da avaliação), e outra referente apenas ao necessário para *download* e atualização do conjunto de instruções referentes à máquina de estados (métrica *M1.4*).

Verificamos na Seção 4.4.2 que o compilador da máquina virtual passou a ser responsável por identificar as configurações válidas para os estados definidos na especificação da máquina de estados através do valor da faixa de objetivos de qualidade definido individualmente para cada estado. Cada configuração válida para a LPSD irá possuir um valor de objetivo de qualidade e a faixa de objetivo de qualidade será usada para selecionar um conjunto destas configurações para cada estado. Assim, para avaliar a métrica *M3.1*, tentamos identificar 10 objetivos de qualidade para os quais existiam apenas uma configuração possível e outros 10 objetivos para os quais não existiam configurações possíveis. Esses valores seriam aplicados dois a dois (um válido e um sem configurações, nesta ordem) em uma especificação de máquina de estados mais simples contendo apenas dois estados e nenhuma referência a outras *features*. O resultado esperado seria uma mensagem do compilador indicando a inexistência de configurações válidas para o segundo estado.

Para esta simulação, limitamos a construção da tabela de configurações utilizando apenas a respectiva confiabilidade das configurações. Entretanto, verificamos que todas

as configurações possíveis possuiam um dentre seis valores distintos de confiabilidade, os quais podem ser verificados na Tabela 5.2. Dada a quantidade de configurações em cada valor, pudemos observar que não seria possível indicar objetivos de qualidade que possuíssem apenas uma configuração. A Tabela 5.3 apresenta seis objetivos de qualidade mínimos (considerando objetivos de qualidade informados com apenas uma casa decimal) para as quais existem configurações válidas, sendo consequentemente invalidas quaisquer outras faixas que não contenham em si as faixas válidas. O teste foi realizado utilizando estas faixas de objetivos de qualidade e o código da máquina de estados foi processado pelo compilador da DSL.

Confiabilidade	Nº de configurações
0.914938993906424756	4
2.623705881827839222	32
4.347928849045083322	99
6.087747698353311618	162
7.843303497087097690	141
9.614738588558402452	8

Tabela 5.2: Distribuição de configurações por objetivo de qualidade.

Objetivos de qualidade	
Válidos (mínimos)	Sem configurações
	$0.0 < x \leq 0.9$
$0.9 < x \leq 1.0$	$1.0 < x \leq 2.6$
$2.6 < x \leq 2.7$	$2.7 < x \leq 4.3$
$4.3 < x \leq 4.4$	$4.4 < x \leq 6.0$
$6.0 < x \leq 6.1$	$6.1 < x \leq 7.8$
$7.8 < x \leq 7.9$	$7.9 < x \leq 9.6$
$9.6 < x \leq 9.7$	$9.7 < x \leq 10$

Tabela 5.3: Faixas de objetivo de qualidade válidos e inválidos.

É importante destacar que o resultado dessa avaliação é dependente da tabela de configurações e objetivos de qualidade utilizada pelo servidor. O uso de outros parâmetros ou fórmulas para a definição de uma tabela diferente irá produzir resultados diferentes daqueles aqui observados, uma vez que serão alterados os objetivos de qualidade de cada configuração válida. Com base neste princípio, e com o objetivo de garantir o funcionamento do sistema usando diferentes tabelas de configurações, optamos por realizar um segundo teste semelhante ao anterior, este usando uma tabela criada especificamente para esta simulação. Assim, considerando os valores de objetivo de qualidade válidos na DSL (valores entre 0 e 10 inclusive), construímos uma nova tabela de configurções e objetivos de qualidade dividindo os valores válidos em 20 faixas sequenciais com intervalos de 0.5

para cada faixa; em apenas metade dessas faixas adicionamos uma única configuração aleatória. O teste anteriormente descrito para a métrica *M3.1* foi então repetido substituindo a tabela original por esta.

Todos os testes do sistema Monitor foram realizados usando um *smartphone* Samsung Galaxy S3 Mini, com 8GB de armazenamento interno, processador de 1.0GHz dual-core, 1GB de memória RAM, sistema operacional Andriod 4.1.2 (Jelly Beans) e 1 ano de comprado. O referido aparelho não é usado regularmente como celular e não tem outros aplicativos em execução simultânea, exceto aqueles requeridos pelo sistema operacional.

5.3 Resultados e Análise

Conforme cenários e contexto indicados na seção anterior, procedemos à coleta e análise de dados como descrito a seguir. Os dados foram coletados na forma de um *log* unificado, e foi feita a separação e organização dos dados em uma planilha para análise. Os resultados estão sintetizados na Tabela 5.4 e são discutidos nas subseções seguintes.

Métrica	Média	Desvio-padrão
Tempos de resposta (ms)		
M1.1	3.53	± 16.01
M1.2	43.52	± 56.63
M1.3	4508.59	± 3215.49
M1.4	1366.59	± 886.15
Consumo de recursos		
M2.1	108.82 horas	± 6.93
M2.2	5902.20 kb	± 79.51
M2.3	3097 bytes	-
Safey		
M3.1	Sim	-

Tabela 5.4: Sumarização dos dados.

5.3.1 Tempos de Resposta

Após o primeiro ciclo de carga e descarga para a coleta de dados de avaliação do sistema Monitor, detectamos que foram necessárias aproximadamente 94 horas para que a bateria atingisse a marca de 1% restante. Por não termos julgado necessário coletar data e hora de cada evento no dispositivo neste primeiro momento, optamos por descartar essa medição de tempo para descarga da bateria e conduzir apenas quatro novos ciclos de carga e descarga para obtenção da métrica *M2.1* (que será discutida mais adiante) e

utilizar apenas os testes da primeira medição para as demais métricas, uma vez que a ausência de data e hora de cada evento não influencia os demais resultados.

Durante o periodo desta primeira medição, verificamos que a máquina de estados foi executada 4737 vezes (métrica *M1.1*) com uma média de pouco mais de 3.5 milissegundos para avaliar o estados de saúde a partir dos dados recebidos dos sensores. Das execuções da máquina de estados, apenas 385 resultaram em uma transição de estado (métrica *M1.2*), para a qual apuramos uma média de pouco mais que 43.5 milissegundos. É importante lembrar que este valor também inclui os valores da métrica *M1.1*.

Sobre as métricas *M1.3* e *M1.4*, referentes à atualização da máquina de estados por atuação do especialista no domínio, realizamos a operação de atualização 34 vezes para a coleta dos tempos. Verificamos que foi necessário em média pouco mais que 4.5 segundos para a recepção da notificação de atualização (métrica *M1.3*) e menos de 1.4 segundos em média para o *download* e atualização da máquina de estados a partir da recepção da notificação (métrica *M1.4*). Esta diferença de tempos se justifica pelo fato de o dispositivo executando o programa do Monitor estar na mesma rede onde se encontra o servidor.

Entretanto, observamos uma variação apreciável dos tempos para estas métricas relacionadas à questão *Q1*, indicado pelos respectivos desvios-padrão dessas métricas. Esta variação pode ter ocorrido devido a interferências externas ao aplicativo, tais como escalonamento do sistema operacional, alocação de memória, atividade de rede, etc. Ao analisar os valores obtidos para as métricas *M1.1* e *M1.2* nesta medição, identificamos que, em alguns casos, houveram valores muito altos de tempos de execução. Realizamos uma análise mais apurada dos dados obtidos, bem como uma comparação com os dados obtidos durante a segunda execução, através da criação de gráficos do tipo *boxplot*. A Figura 5.1 apresenta estes gráficos para o primeiro (a) e segundo (b) ciclos de carga e descarga da bateria. Os gráficos foram reduzidos para representar somente o valor máximo de 60 milissegundos para permitir sua melhor visualização, dados os altos valores máximos em ambos os ciclos.

Ao analisarmos e compararmos estes gráficos, observamos que a os tempos de execução na mediana, tanto da máquina de estados (*M1.1*) quanto da reconfiguração completa da LPSD (*M1.2*), estão concentrados em uma faixa menor de distribuição dos valores e seus terceiros quartis se apresentam a boa distância dos valores máximos observados (482.26 para *M1.1* e 633.60 para *M1.2* no primeiro ciclo e 1393.91 para *M1.1* e 795.03 para *M1.2* no segundo ciclo). Assim, podemos afirmar que os altos tempos observados ocorreram esporadicamente ao longo da simulação.

O mesmo gráfico para análise foi feito sobre os dados de atualização da máquina de estados, cujo resultado pode ser visto na Figura 5.2, onde foram analisados os tempos de recepção da notificação de atualização disponível (*M1.3*) e de conclusão do processo

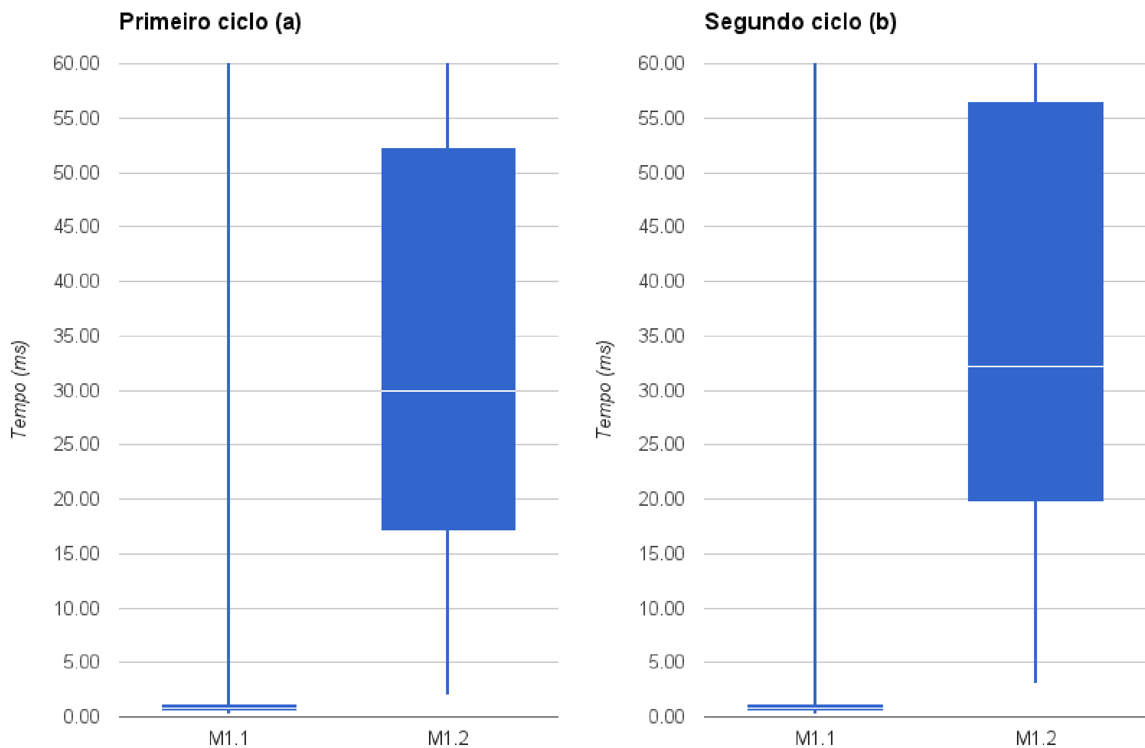

Figura 5.1: Comparação de tempos de execução em dois ciclos de carga e descarga

de atualização (*M1.4*). Neste gráfico, os tempos estão em segundos e também podemos observar uma distribuição uniforme dos tempos de atualização.

Apesar desta análise, procuramos estudar os motivos responsáveis pelos tempos mais elevados. Nossa primeira linha de investigação foi consumo do processador, dado que o consumo de memória observado é uniforme (o consumo de memória será discutido na próxima seção). Utilizamos uma ferramenta disponível como parte do *kit* de desenvolvimento Android dentro da ferramenta Eclipse[3] para analisar o consumo do processador. A Figura 5.3 demonstra uma distribuição média do consumo do processador com o aplicativo do sistema Monitor em execução sem coleta de dados (a) e com a coleta de dados (b).

No primeiro gráfico, o aplicativo Monitor sequer é listado ocupando algum percentual do processador, demonstrando que sua intrface gráfica não exige esforço significativo do processador para execução, enquanto no segundo gráfico podemos observar, alguns segundos após o início da simulação para coleta de dados, que o aplicativo Monitor é o principal consumidor do processador, com partes do processo executando no nível do *kernel* do sistema e no espaço do usuário. Com base neste achado, pudemos identificar dois fatores que podem ter contribuído para a obtenção dos tempos de execução e atualização mais elevados: (1) a ativação periódica da tela, realizada em intervalos irregulares para verificar se o ciclo de descarga havia sido concluído, e (2) a intervenção de outros aplicativos, tais

[3]http://www.eclipse.org

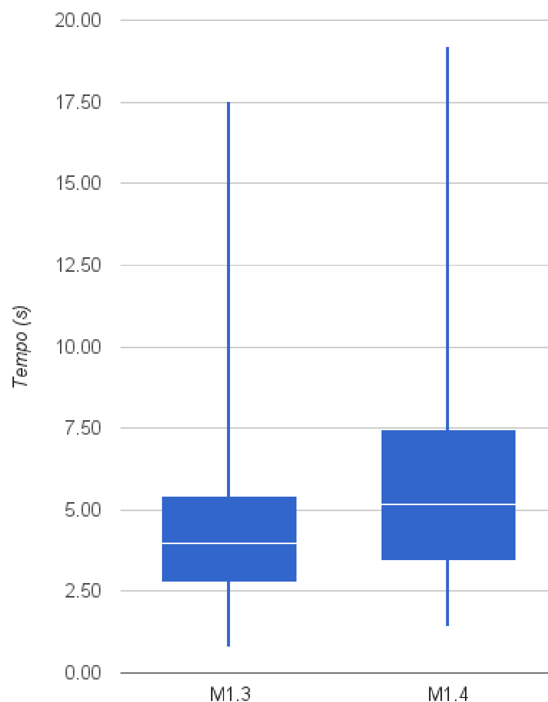

Figura 5.2: Gráfico de análise dos tempos de atualização

como atualização de software através de rede sem fio, cuja conexão foi mantida ativa após a coleta de dados para a métrica *M1.3*, bem como do software de anti-vírus instalado, executado após o *download* de cada atualização.

Desta forma, procedemos à realização uma nova simulação, desta vez configurando o dispositivo móvel para modo avião, assim desativando toda conectividade do aparelho. A Figura 5.4 apresenta a distribuição dos valores coletados para as métricas *M1.1* e *M1.2* ao longo desta nova simulação.

Nesta nova simulação, o aplicativo foi deixado em execução por apenas 5 dias, tendo em vista que neste periodo apenas 31% de sua bateria havia sido descarregada. Ao fim deste período e repetindo o processo de análise dos dados, observamos que os tempos máximos de execução foram reduzidos significativamente. Entretanto, pudemos também observar um aumento geral nos tempos de execução, em especial o tempo de execução da máquina de estados (*M1.1*).

Assim, optamos por concluir que as interrupções e operações produzidas pelo sistema operacional em plano de fundo em diferentes modos de utilização do dispositivo móvel são variáveis espúrias que fogem ao nosso controle para a realização das simulações e que podem produzir efeitos adversos nos tempos de execução monitorados. Entretanto também pudemos observar que, na maior parte dos casos, os tempos de execução permanecem

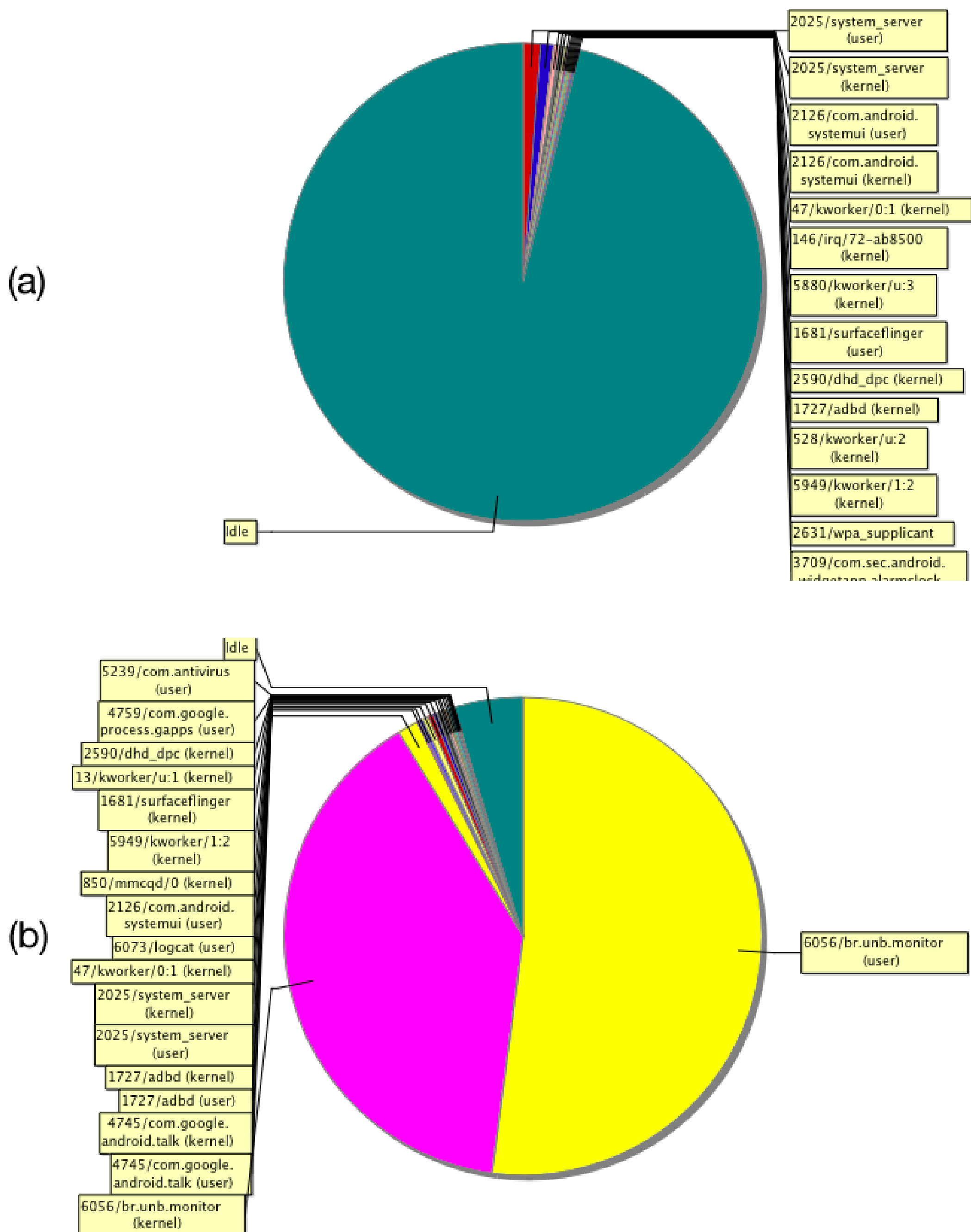

Figura 5.3: Tempo de ocupação do processador pelo sistema sem coleta de dados (a) e com coleta de dados (b)

dentro de um limite que consideramos satisfatório para uma aplicação deste domínio.

5.3.2 Consumo de Recursos

Como destacado no início da seção anterior, optamos por reduzir a quantidade de execuções até exaustão da bateria para um total de apenas quatro devido ao alto tempo que

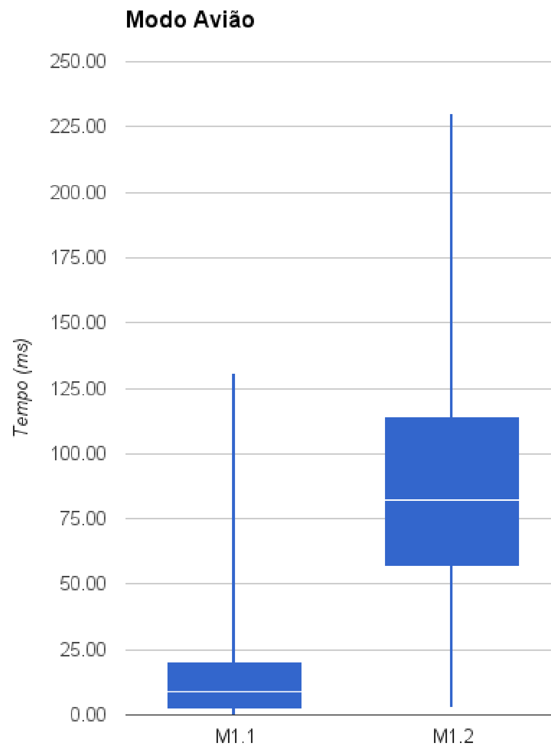

Figura 5.4: Gráfico de análise dos tempos de execução em modo avião

registramos durante a primeira execução. Para estas execuções, foram feitas adaptações no mecanismo de coleta de dados para permitir obter a hora exata de todos os registros de modo a nos permitir usar a data do último registro no *log* como a hora mais aproximada da exaustão da bateria. A Tabela 5.5 detalha os valores obtidos durante esta avaliação indicando o tempo gasto em horas para que a bateria atingisse 50% e 100% de descarga.

	Consumo da bateria	
Ciclo	50% (horas)	100% (horas)
1	54.17	102.50
2	58.97	107.25
3	61.21	118.70
4	56.62	106.83
Média	57.74 ± 3.03	108.82 ± 6.93

Tabela 5.5: Tempos para exaustão da bateria

Como podemos observar, há uma relação linear de descarga da bateria, a qual é justificada devido a não estarmos utilizando sensores reais, mas simulando através de dados em arquivos. Em todos os demais ciclos de carga e descarga da bateria, o tempo necessário até a descarga completa excedeu 4 dias (96 horas). Julgamos que este resultado foi obtido pela ausência de um *SIM card* que habilitasse as funções de telefonia e optamos por realizar um teste adicional identico aos anteriores, porém acrescentando um *SIM card*

habilitado ao dispositivo. Os tempos obtidos de 43.05 horas para exaustão de 50% e de aproximadamente 80.27 horas para exaustão completa, confirmam parcialmente nossa teoria, uma vez houve uma redução de aproximadamente 20% na autonomia durante esta simulação em relação à média da simulações anteriores. Na execução em modo avião (vide Figura 5.4), observamos uma descarga de 31% da bateria em um período de 107 horas, após as quais optamos por não dar continuidade a esta simulação, mas estimamos que a descarga completa da bateria em modo avião se daria em aproximadamente 345.16 horas, ou seja, em 14.38 dias, com um ganho aproximado de 200% na sua autonomia.

Concorrentemente à coleta dos tempos, foi realizada periodicamente a coleta de dados referentes ao consumo de memória da aplicação para avaliação da métrica *M2.2*. Durante o período avaliado, o consumo de memória foi coletado 293 vezes, das quais obtivemos uma média de pouco mais de 5.7 MB alocados exclusivamente para o programa Monitor. Esta memória, como citada anteriormente, é alocada exclusivamente para o programa e não pode ser paginada, sendo portanto memória exclusiva da aplicação.

Data o grande número de elementos variáveis que compõem o arquivo da máquina de estados (número de estados, propriedades dos estados, condições de mudança de estado, tabela de configurações, etc.), optamos por medir somente o tamanho da máquina de estados de exemplo. Sobre esta, o tamanho de 3097 bytes pode ser considerado pequeno e é um dos fatores que influencia o tempo de resposta para a métrica *M1.3b*.

5.3.3 *Safety*

A avaliação de *safety* foi realizada de forma independente das demais avaliações e sem transmissão da máquina de estados para o dispositivo móvel, uma vez que a coleta de dados para esta simulação depende apenas do compilador presente no servidor de aplicação, conforme descrito na Seção 4.4.2. Os dois testes descritos para a métrica *M3.1* foram realizados, nos quais obtivemos satisfatoriamente os resultados esperados, conforme indicado na Tabela 5.4.

5.4 Ameaças à Validade

De modo a realizar a avaliação descrita neste capítulo, algumas variáveis precisaram ser controladas e/ou restritas. Este controle pode ter trazido consigo algumas ameaças à validade da avaliação, as quais reconhecemos abaixo.

- *Validade de construção*:
  - Para avaliar a capacidade da técnica proposta por este trabalho, foi feita apenas uma aplicação sobre o trabalho de Fernandes [2012];

  - Da mesma forma, as simulações foram realizadas utilizando apenas o mesmo conjunto de três estados de risco de saúde (normal, risco moderado e risco alto) da LPSD original apesar de a implementação desta proposta permitir a definição de até 256 estados;

- *Validade interna*:
  - A coleta dos dados foi feita através de chamadas dentro do aplicativo móvel e os armazenados em arquivo no próprio dispositivo. Estas chamadas não devem trazer impacto significativo para a execução das funções avaliadas;
  - Foram usados dados simulados a partir de um arquivo dentro do próprio dispositivo. Conforme explicado no início deste capítulo, esse método de simulação foi utilizado devido à dificuldade em observar as diversas condições que disparam transições em indivíduos reais. A simulação também foi feita sem a transmissão de dados através de uma conexão *Bluetooth*, o que poderia causar uma redução no tempo de duração da bateria;

- *Validade externa*:
  - O dispositivo não era utilizado cotidianamente, ou seja, durante os testes, o dispostivo não foi utilizado com outras funções e mesmo sem um *SIM card* (funções de telefonia). Apesar da realização de um ciclo de descarga da bateria com o uso de um *SIM card*, o uso exclusivo do dispositivo para a avaliação pode ter contribuído para alta duração da bateria e para os tempos de execução da máquina virtual, de reconfiguração e de atualização da máquina de estados;
  - Igualmente, a presença do dispositivo celular e do servidor na mesma rede influencia o resultado do tempo para atualização da máquina de estados. Reconhecemos que, em condições normais, ambos os aparelhos devem se encontrar em redes diferentes cuja distância real é variável, bem como a probabilidade desta atualização ser feita através de redes celulares de capacidade variável, de acordo com a localização.
  - As simulações de atualização foram realizadas utilizando apenas um dispositivo e, consequentemente, não foram realizadas simulações de atualização utilizando diversos usuários simultaneamente.
  - A tabela de configurações pode ser construída pois a quantidade de configurações válidas para a LPSD avaliada é relativamente pequena e gerenciável e isto levou a uma redução no tempo para seleção de uma configuração. Alterações

no modelo de *features* da linha de produtos ou na estratégia usada para determinação do objetivo de qualidade das configurações podem tornar o uso desta tabela inviável.

- *Confiabilidade*:
  - O estudo através de simulação foi implementado de forma a permitir sua reprodução com resultados similares. Diferenças de resultado na simulação são esperadas com o uso de dispositivos e configurações de rede diferentes.

Considerando o sistema Monitor aqui avaliado um projeto em andamento e que nem todas as funções necessárias para a realização completa das avaliações descritas nesta seção estão disponíveis, recomenda-se que esta avaliação seja revisada em futuras atualizações do sistema.

5.5 Resumo

Com a execução da avaliação descrita neste capítulo, pudemos verificar que o modelo de evolução da linha de produtos proposto neste trabalho manteve o comportamento esperado do sistema Monitor, ou seja, a seleção de configurações de maior ou menor qualidade foi realizada de acordo com o respectivo aumento ou redução do risco de saúde simulado.

Ainda podemos considerar que a introdução de uma máquina virtual para a execução das instruções referentes à máquina de estados não compromete o tempo de resposta do sistema, tendo em vista que o tempo médio de execução da mesma não chega a 10% do tempo total médio necessário para a reconfiguração completa do sistema. Em particular, este tempo sugere que o uso da máquina virtual não compromete o tempo para reconfiguração da linha de produtos mesmo em um dispositivo móvel e de recursos limitados. O consumo de memória pela aplicação também demonstrou ser relativamente pequeno e estável, ocupando apenas cerca de 1% da memória mínima suportada pelo sistema operacional atualmente, que é de 512 MB [Google, 2014a].

Assim, a avaliação sugere que o desempenho da LPSD com suporte a evolução de objetivos é aceitável por manter o comportamento esperado da mesma com um *footprint* baixo, e a propriedade de *safety* é alcançada. No entanto, existem algumas ameaças à validade e futuros estudos precisam ser conduzidos para aumentar a evidência dos resultados obtidos.

Capítulo 6

Conclusão

Em extensão ao conceito de linha de produtos de software (LPS), uma linha de produtos de software dinâmica (LPSD) posterga a resolução de variabilidades para o momento da execução do software, permitindo que este altere sua configuração de acordo com a necessidade de resposta a alterações no seu contexto. Apesar de inúmeros trabalhos lidarem com a evolução de LPSDs, verificamos estar em aberto a questão relativa à evolução dos objetivos da LPSD em tempo de execução, bem como a possibilidade desta evolução ser diferente por instância da LPSD.

Para resolver o problema em vista, elaboramos um modelo para evolução individual de objetivos para LPSDs cientes de qualidade (MEIOq), consistindo de um processo para desenvolvimento deste tipo de LPSD com características que permitem a um especialista no domínio especificar os objetivos da LPSD em uma linguagem específica para o domínio da aplicação. Foi apresentada uma arquitetura complementar contemplando meios para a especificação e atualização individual destes objetivos e sem a presença do dispositivo onde a LPSD esteja em execução. Como forma de otimizar o desempenho da solução, foi proposto o uso de uma pequena máquina virtual, para a qual também foi construído um compilador para realizar a transformação entre a linguagem do especialista no domínio em uma linguagem com instruções imperativas que podem ser executadas diretamente pela LPSD.

Foi feita uma avaliação da metodologia proposta através da prova de conceito via implementação da mesma em uma LPSD para monitoração de sinais vitais de um indivíduo através de uma rede de sensores do corpo humano e da avaliação sistemática, através de simulação com dados de sinais vitais, da solução implementada para verificar a manutenção do comportamento original da LPSD com a implementação da proposta. A simulação utilizada para validar a implementação foi construída de forma a permitir sua reprodução com outros cenários, bem como a verificação e validação de trabalhos futuros.

6.1 Limitações

A prova de conceito realizada através da implementação do MEIOq considerou somente uma linha de produtos de software dinâmica fechada, ou seja, uma linha de produtos cujas variantes possíveis para cada ponto de variação são conhecidas em tempo de compilação. Trabalhos futuros podem avaliar a adequação do MEIOq para LPSDs abertas.

Da mesma forma, para realizar a seleção de configurações válidas para um estado, este trabalho considerou somente um parâmetro de qualidade, no caso a confiabilidade da configuração. É importante destacar que a confiablidade é apenas um dos atributos da dependabilidade [Avizienis et al., 2004] e que outros parâmetros podem ser utilizados nesta seleção. O trabalho de Fernandes [2012] também usou duas diferentes estratégias para o cálculo do objetivo de qualidade de uma configuração, se utilizando diferentes parâmetros de qualidade para isso.

Adicionalmente, a validade da avaliação pode ser ameaçada. Considerando a validade interna, são consideradas ameaças a ausência de testes com sujeitos e sensores reais, devido à necessidade de controle de variáveis, e o uso de dados simulados. Da mesma forma, ameaçam a validade externa o uso do dispositivo móvel exclusivamente para o teste, o compartilhamento da rede para comunicação entre o dispositivo e o servidor de aplicações e a construção da tabela de configurações.

6.2 Trabalhos Relacionados

Nesta seção, buscamos identificar trabalhos relacionados à evolução de linhas de produtos de software dinâmicas, em especial LPSDs orientadas a objetivos. O restante desta seção irá apresentar tais trabalhos, suas limitações no tratamento da evolução de objetivos, os diferenciando do trabalho atual.

Em seu trabalho, Fernandes [2012] definiu um método para construção de LPSDs capazes de selecionar configurações adequadas com base em múltiplos atributos de qualidade e realizou a implementação da solução proposta na forma de um aplicativo móvel para plataforma Android, o qual foi também utilizado no estudo realizado no presente trabalho. Em sua estruturação, Fernandes [2012] definiu uma linguagem para estabelecimento de regras para identificação do estado de saúde de um indivíduo (o objetivo da LPSD), mas estas ainda deveriam ser transformadas manualmente em código a ser incorporado na aplicação em tempo de compilação. O presente trabalho estendeu o trabalho da autora permitindo que os objetivos da LPSD fossem definidos em tempo de execução, além de individualizados de acordo com as necessidades do especialista no domínio para cada indivíduo a ser monitorado.

Sadilek [2008] propõe a criação de uma linguagem específica de domínio para a programação de dispositivos micro-controlados chamados *Wireless sensos network* (WSN) *nodes*, o que é originalmente feito em C ou Assembler dificultando a programação por um especialista no domínio. Existem muitas semelhanças entre este e o corrente trabalho, inclusive a definição de um modelo para especificação de uma linguagem específica do domínio (DSL) e o uso de uma máquina virtual (VM) simplificada para execução de construções feitas utilizando a DSL em um WSN *node*. O autor cita a criação de um ambiente para simulação de construções da DSL e não especifica um modelo para distribuição dos objetivos alterados dinamicamente, o que pode sugerir uma lacuna no trabalho ou a necessidade de atualização individual manualmente de cada nó. Da mesma forma, o trabalho proposto não lida com a reconfiguração automática da rede de sensores, apenas com as regras para detecção de eventos. O trabalho publicado, entretanto, é apresentado como preliminar e não foram encontradas publicações a cerca da conclusão deste trabalho.

Ghezzi e Sharifloo [2010] discutem como LPSDs podem se adaptar em tempo de execução a mudanças no ambiente que causem violações nos requisitos, tais como mudanças na infraestrutura, parâmetros de qualidade de serviço, componentes integrados ou perfis de utilização, de modo que o sistema se mantenha adequado aos requisitos de forma que a implementação em execução produza o mínimo de perturbações. Os autores descrevem, entretanto, uma noção pouco convencional para o produto da LPSD, no sentido de não se referirem a código em execução mas a modelos de mais alto nível; a forma como estes modelos são transformados em implementações e distribuídas foi ignorada no trabalho dos autores.

Da mesma forma, Welsh et al. [2011] apresenta um método (REAssuRE) para substituir a estratégia usada para atingir os objetivos de um sistema ao perceber alterações no contexto de execução. Esta proposta, entretanto, se baseia na modelagem de objetivos de agentes computacionais e não-computacionais no domínio e em uma extensão de um raciocinador i* para determinar a melhor estatégia para satisfação dos objetivos.

Calinescu et al. [2011] propôs um *framework* para a realização prática de uma arquitetura para sistemas baseados em serviços adaptativa através de dois mecanismos complementares: o primeiro é responsável pela seleção dinâmica dos serviços que compõem o sistema com base em características de qualidade requeridas e disponíveis pelos seviços, enquanto o segundo componente ajusta recursos alocados para os serviços individualmente dinamicamente. O trabalho proposto pelos autores não é para explicitamente uma LPSD, apesar de apresentar características como variabilidade (dos serviços) e sua resolução em tempo de execução. A seleção dos serviços a serem utilizados (configuração) do sistema é feita com base nos objetivos de qualidade definidos para o sistema e os serviços, e a intervenção de um especialista no domínio pode ser necessária caso o sistema não encontre

um serviço adequado, mas os autores não se preocuparam com a alteração dos objetivos do sistema em tempo de execução.

Hallsteinsen et al. [2012] apresentam um *framework* e metodologia orientada por modelos para facilitar o projeto e implementação de aplicações adaptativas sensíveis ao contexto, denominado MUSIC. A solução proposta suporta uma variedade de mecanismos de adaptação, tais como parâmetros de configuração, substituição de ligações de componentes e serviços, e redistribuição de componentes numa arquitetura computacional distribuída, permitindo a integração de serviços externos e seus parâmetros de qualidade no gerenciamento de adaptação em tempo de execução de forma totalmente autônoma. Este trabalho está relacionado com a variabilidade aberta de LPSDs, sem lidar com interferências por um especialista no domínio. Além disso, o objetivo do sistema é dado por cada componente dentro do ambiente computacional. Adicionalmente, os autores não tratam explicitamente do gerenciamento da variabilidade dos objetivos de qualidade, seja no tempo (evolução) ou no espaço (individualização), conforme em nossa pesquisa.

De forma semelhante a Hallsteinsen et al. [2012], Ortiz et al. [2012] apresenta um modelo para variação estrutural de LPSDs focado na adição, remoção e substituição de variantes em tempo de execução. Os autores propõem a noção de supertipos em sistemas de *features* para capturar a essência dos pontos de variação e permitir o agrupamento de variantes sob um supertipo compatível. A resolução de variabilidade é realizada observando este supertipo em cada variante para reconstruir o modelo de *features*. Os autores também não abordam a capacidade de evolução ou individualização de objetivos e, ao contrário deste trabalho, nossa abordagem proposta é baseada no uso de um modelo de *features* fechado.

Capilla et al. [2014] procuram identificar uma visão geral do estado da arte e técnicas atuais que tentam resolver desafios de mecanismos de variabilidade em tempo de execução no contexto de LPSDs. Os autores descrevem que, baseado nestes desafios, o processo de desenvolvimento de LPSDs deve incluir atividades que suportem sua natureza de tempo de execução. Este processo (vide Figura 2.2) apresenta grande semelhança com o processo proposto para o MEIOq para a engenharia de domínio; as atividades da engenharia da aplicação na nossa abordagem são realizadas pela própria LPSD utilizando MAPE. Adicionalmente, os autores descrevem um conjunto de desafios e soluções comuns a LPSDs, entretanto aborda apenas a questão suporte a novas variantes na LPSD sem tratar explicitamente dos objetivos.

6.3 Trabalhos Futuros

Devido às limitações do trabalho proposto, algumas questões de pesquisa podem ser exploradas em trabalhos futuros.

- **Realização de estudos empíricos usando sensores reais**: A simulação realizada no Capítulo 5 foi feita utilizando dados simulados lidos de um arquivo presente no próprio dispositivo, para permitir a observação do comportamento do sistema em um contexto controlado. Novos estudos usando sensores reais devem ser realizados para verificar a adequação da arquitetura e da implementação a cenários de uso cotidianos.

- **Utilização de outras estratégias para cálculo da qualidade de configurações**: Para simplificar este estudo, a análise de qualidade das configurações foi reduzida à identificação e uso da confiabilidade. Outras estratégias, tais como as utilizadas por Fernandes [2012], podem ser incorporadas ao modelo para verificar sua adequabilidade e confirmar o ganho em desempenho obtido com a adoção da metodologia proposta.

- **Adoção de um modelo de *features* aberto**: A construção de uma tabela com parâmetros de qualidade das configurações da LPSD só foi possível pois a quantidade de configurações possíveis na LPSD usada é pequena. Ao se adotar um modelo de *features* mais aberto e flexível, permitindo a adição de novas *features* em tempo de execução, deve ser utilizada uma nova estratégia para construção da tabela de configurações.

- **Verificação de adequação do modelo proposto a outros domínios**: A proposta deste trabalho foi implementada para uma LPSD específica e previamente desenvolvida. Aplicar a metodologia proposta para desenvolvimento de outras LPSDs, desde o início ou como evolução da linha de produtos, é interessante para verificar a adequação do modelo a outros domínios ou identificar ajustes que se façam necessários para obter esta adequação.

Referências

Michail Anastasopoulos. Evolution Control for Software Product Lines: An Automation Layer over Configuration Management. Tese de Doutorado, Instituto Fraunhofer, Stuttgart, Alemanha, 2013. 30

Michalis Anastasopoulos e Cristina Gacek. Implementing product line variabilities. Em Proceedings of the Symposium on Software Reusability: Putting Software Reuse in Context, pp. 109–117, 2001. 6

Algirdas Avizienis, J-C Laprie, Brian Randell, e Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. Transactions on Dependable and Secure Computing, 1(1):11–33, 2004. 3, 68

Victor R. Basili, Gianluigi Caldiera, e H. Dieter Rombach. The goal question metric approach. Em Encyclopedia of Software Engineering. Wiley, 1994. 52

Nelly Bencomo, Gordon S. Blair, Carlos A. Flores-Cortés, e Peter Sawyer. Reflective component-based technologies to support dynamic variability. Em Proceedings of the Second International Workshop on Variability Modelling of Software-Intensive Systems, pp. 141–150, 2008. 19

Nelly Bencomo, Svein Hallsteinsen, e Eduardo S. Almeida. A view of the dynamic software product line landscape. Computer, 45(10):36–41, 2012. x, 7, 8, 9, 18

Radu Calinescu, Lars Grunske, Marta Z. Kwiatkowska, Raffaela Mirandola, e Giordano Tamburrelli. Dynamic qos management and optimization in service-based systems. IEEE Trans. Software Eng., 37(3):387–409, 2011. 69

Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz Cortés, e Mike Hinchey. An overview of dynamic software product line architectures and techniques: Observations from research and industry. Journal of Systems and Software, 91(0):3–23, 2014. x, 7, 8, 9, 70

Carlos Cetina, Pau Giner, Joan Fons, e Vicente Pelechano. Designing and prototyping dynamic software product lines: techniques and guidelines. Em Proceedings of the 14th International Conference on Software Product Lines (SPLC'10), pp. 331–345. IEEE, 2010. 8

Paul Clements e Linda Northrop. Software Product Lines: Practices and Patterns. Addison Wesley Professional, Boston, 2002. 5

Ian D. Craig. Virtual Machines. Springer, London, 2006. 15, 16, 17, 41

Krzysztof Czarnecki e Ulrich W. Eisenecker. Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., New York, 2000. x, 5, 6, 7, 22, 23

Paula Gabriela de Medeiros Fernandes. Linha de produtos de software dinâmica direcionada por qualidade: O caso de redes de monitoração do corpo humano. Dissertação de Mestrado, Departamento de Ciência da Computação, UnB - Universidade de Brasília, Brasília, 2012. x, 3, 10, 11, 12, 19, 23, 25, 35, 36, 38, 46, 52, 55, 64, 68, 71

Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides. Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley Professional, Boston, 1994. 41, 50

David Garlan, Felix Bachmann, James Ivers, Judith Stafford, Len Bass, Paul Clements, e Paulo Merson. Documenting Software Architectures: Views and Beyond. Addison-Wesley Professional, Boston, 2nd edition, 2010. 38

Carlo Ghezzi e Amir Molzam Sharifloo. Dealing with non-functional requirements for adaptive systems via dynamic software product-lines. Em Software Engineering for Self-Adaptive Systems, pp. 191–213, 2010. 69

Google. Running Android with Low RAM. `https://source.android.com/devices/low-ram.html`, 2014a. 66

Google. Investigating Your RAM Usage. `https://developer.android.com/tools/debugging/debugging-memory.html`, 2014b. 55

Thomas R. G. Green e Marian Petre. When visual programs are harder to read than textual programs. Em Human-Computer Interaction: Tasks and Organisation, Proceedings of Sixth European Conference on Cognitive Ergonomics, pp. 167–180, 1992. 13, 31

Svein Hallsteinsen, Erlend Stav, Arnor Solberg, e Jacqueline Floch. Using product line techniques to build adaptive systems. Em Proceedings of the 10th International Software Product Line Conference (SPLC'06), pp. 141–150. IEEE, 2006. 1, 7, 18

Svein Hallsteinsen, Mike Hinchey, Sooyong Park, e Klaus Schmid. Dynamic software product lines. Computer, 41(4):93–95, 2008. 1

Svein Hallsteinsen, Kurt Geihs, Nearchos Paspallis, Frank Eliassen, Geir Horn, Jorge Lorenzo, Alessandro Mamelli, e George Angelos Papadopoulos. A development framework and methodology for self-adapting applications in ubiquitous computing environments. Journal of Systems and Software, 85(12):2840–2859, 2012. 7, 70

Yang Hao e Robert Foster. Wireless body sensor networks for health-monitoring applications. Physiological Measurement, 29(11):R27–R56, 2008. 10

Paul Hudak. Modular domain specific languages and tools. Em Proceedings of Fifth International Conference on Software Reuse, pp. 134–142. IEEE, 1998. 12

Jeffrey O. Kephart e David M. Chess. The vision of autonomic computing. Computer, 36(1):41–50, 2003. 8

Charles W. Krueger. Variation management for software production lines. Em Software Product Line, pp. 37–48. Springer, 2002. 2

Jaejoon Lee e Kyo Chul Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. Em Proceedings of the 10th International Software Product Line Conference (SPLC'06), pp. 131–140. IEEE, 2006. 7, 18

Tim Lindholm, Frank Yellin, Gilad Bracha, e Alex Buckley. The java virtual machine specification: Java se 7 edition. Relatório técnico, Oracle, 2013. URL `http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf`. 15

Jeff Magee e Jeff Kramer. Dynamic structure in software architectures. ACM SIGSOFT Software Engineering Notes, 21(6):3–14, 1996. 1

Marjan Mernik, Jan Heering, e Anthony M. Sloane. When and how to develop domain-specific languages. ACM Computing Surveys, 37(4):316–344, 2005. 13, 14, 22, 31, 40

Vinicius Uriel Cardoso Nunes. Gerência de variabilidade em modelos de confiabilidade para linhas de produto de software. Dissertação de Mestrado, Departamento de Ciência da Computação, UnB - Universidade de Brasília, Brasília, 2012. 38

Óscar Ortiz, Ana Belén García, Rafael Capilla, Jan Bosch, e Mike Hinchey. Runtime variability for dynamic reconfiguration in wireless sensor network product lines. Em Proceedings of the 16th International Software Product Line Conference (SPLC'12), volume 2, pp. 143–150. ACM, 2012. 1, 70

Marko Rosenmüller, Norbert Siegmund, Mario Pukall, e Sven Apel. Tailoring dynamic software product lines. ACM SIGPLAN Notices, 47(3):3–12, 2011. 7

Daniel A. Sadilek. Domain-specific languages for wireless sensor networks. Em Modellierung, volume 127 of LNI, pp. 237–241. GI, 2008. 68

Hesham Shokry e M Ali Babar. Dynamic software product line architectures using service-based computing for automotive systems. Em Proceedings of the 12th International Software Product Line Conference (SPLC'08), pp. 53–58. IEEE, 2008. 1

Ian Sommerville. Software Engineering. Addison Wesley Professional, Harlow, Inglaterra, 9th edition, 2010. 1

Mark Strembeck e Uwe Zdun. An approach for the systematic development of domain-specific languages. Software, Practice and Experience, 39(15):1253–1292, 2009. 14, 15, 27, 36

Mikael Svahnberg, Jilles van Gurp, e Jan Bosch. A taxonomy of variability realization techniques. Software, Practice and Experience, 35(8):705–754, 2005. 5, 6, 7

Arie van Deursen, Paul Klint, e Joost Visser. Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices, 35(6):26–36, 2000. 13, 14, 15, 22, 27, 31, 39

Jilles van Gurp, Jan Bosch, e Mikael Svahnberg. On the notion of variability in software product lines. Em Proceedings of the Working IEEE / IFIP Conference on Software Architecture, pp. 45–54, 2001. 7

Yang Wang, Alfred Kobsa, André Van Der Hoek, e Jeffery White. Pla-based runtime dynamism in support of privacy-enhanced web personalization. Em Proceedings of the 10th International Software Product Line Conference (SPLC'06), pp. 151–162. IEEE, 2006. 7, 18

Kristopher Welsh, Pete Sawyer, e Nelly Bencomo. Towards requirements aware systems: Run-time resolution of design-time assumptions. Em International Conference on Automated Software Engineering, pp. 560–563. IEEE, 2011. 69

Reinhard Wilhelm e Helmut Seidl. Compiler Design: Virtual Machines. Springer, Berlin, 2010. 15, 17

Reinhard Wolfinger, Stephan Reiter, Deepak Dhungana, Paul Grunbacher, e Herbert Prahofer. Supporting runtime system adaptation through product line engineering and plug-in techniques. Em Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008), pp. 21–30. IEEE, 2008. 1