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

Gerência de Variabilidade em Modelos de Confiabilidade para Linha de Produtos de Software

Vinicius Uriel Cardoso Nunes

Brasília
2012

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

Gerência de Variabilidade em Modelos de Confiabilidade para Linha de Produtos de Software

Vinicius Uriel Cardoso Nunes

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Computação

Orientador
Prof. Dr. Vander Alves

Coorientadora
Prof.ª Dr.ª Genaína Rodrigues

Brasília
2012

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

Coordenadora: Prof.ª Dr.ª Mylène Farias

Banca examinadora composta por:

Prof. Dr. Vander Alves (Orientador) — CIC/UnB
Prof. Dr. Rohit Gheyi — UFCG
Prof. Dr. Anderson Nascimento — ENE/UnB

CIP — Catalogação Internacional na Publicação

Nunes, Vinicius Uriel Cardoso.

Gerência de Variabilidade em Modelos de Confiabilidade para Linha de Produtos de Software / Vinicius Uriel Cardoso Nunes. Brasília : UnB, 2012.
99 p. : il. ; 29,5 cm.

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

1. LPS, 2. PARAM, 3. Cadeias de Markov, 4. Model checking

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

Gerência de Variabilidade em Modelos de Confiabilidade para Linha de Produtos de Software

Vinicius Uriel Cardoso Nunes

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Computação

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

Prof. Dr. Rohit Gheyi
UFCG

Prof. Dr. Anderson Nascimento
ENE/UnB

Prof.ª Dr.ª Mylène Farias
Coordenadora do Mestrado em Computação

Brasília, 10 de Dezembro de 2012

Dedicatória

À minha família.

Agradecimentos

Agradeço a todos que de alguma forma contribuíram para a realização desse trabalho: colegas de trabalho, colegas de mestrado, professores, parentes e amigos de todas as horas.

Em especial, agradeço ao meu orientador Vander com quem venho trabalhando desde a graduação.

Agradeço à minha coorientadora Genaína por todas as contribuições para o desenvolvimento do tema.

Por fim, agradeço à minha namorada Paula por todo companheirismo e cumplicidade.

Resumo

Software está cada vez mais presente em nosso dia a dia. Em alguns domínios, especialmente os de sistemas críticos, software confiável é uma necessidade. Assegurar confiabilidade não é um problema trivial. *Model checking* pode ser utilizado para estimar a confiabilidade de um software através de modelos que representam a comportamento do sistema. Através destes modelos é possível estimar e medir quantitativamente propriedades como confiabilidade. No contexto das Linhas de produto de software (LPS), é preciso verificar uma família inteira de sistemas. Não é viável construir um modelo para cada configuração de uma LPS uma vez que o número de modelos requerido pode ser muito grande. Algumas contribuições tratam diretamente esta questão propondo técnicas específicas para LPS. Em particular, a técnica de *model checking* paramétrico permite a utilização de uma único modelo para obter valores de propriedades de diferentes configurações através de uma fórmula aritmética. No entanto, mesmo uma fórmula aritmética pode não ser fácil de avaliar em alguns cenários. As técnicas atuais impõem limitações sobre a variabilidade. Lidar com variabilidade por meio de *model checking* paramétrico é ainda um problema em aberto. Nesse trabalho, esse problema é tratado por meio de uma proposta de modelagem para *model checking* paramétrico capaz de representar qualquer tipo de variabilidade. Além disso, apresentamos uma extensão para abordagem proposta capaz de reduzir o tamanho da fórmula paramétrica.

Palavras-chave: LPS, PARAM, Cadeias de Markov, Model checking

Abstract

Software is increasingly present in our daily lives. In some domains, specially those of critical systems, dependable software is a must. Ensuring dependability is not a trivial problem. Model checking can be used to estimate the reliability of a software through models that represent the behavior of the system. Through these models it is possible to estimate and measure quantitatively properties such as reliability. In the context of Software Product Lines (SPL), we need to check an entire family of systems. It is not feasible to build a model for each configuration of a SPL as the number of models required can be very large. Some contributions directly address this issue proposing techniques specifically tailored for SPL. Particularly, the technique of *parametric model-checking* allows the use of a single model to obtain properties values from different configurations through an arithmetic formula. However, even an arithmetic formula may not be easy to evaluate in some scenarios. Current techniques may impose limitations over the variability. To handle variability on parametric model checking is still an open problem. This work addresses this problem by proposing a parametric model checking approach able to represent any type of variability and providing a theoretical basis where this work is grounded. Additionally, we present and extension to this approach able to reduce the size of the parametric formula.

Keywords: SPL, PARAM, Markov Chain, Model checking

Sumário

Lista de Figuras x

Lista de Tabelas xii

1 Introdução 1
1.1 Problema 2
1.2 Solução Proposta 3
1.3 Contribuições 4
1.4 Organização do Trabalho 4

2 Fundamentação Teórica 6
2.1 Análise de Dependabilidade 6
2.2 *Model Checking* Probabilístico 7
2.2.1 Cadeias de Markov 7
2.2.2 Lógica Temporal 11
2.2.3 PRISM 13
2.2.4 Linguagem PRISM 13
2.2.5 PARAM 14
2.3 Linha de Produtos de Software 15
2.3.1 Número de configurações 17

3 Gerência de Variabilidade de Modelos de Confiabilidade em Linhas de Produtos de Software: uma Análise de Escalabilidade e Expressividade 18
3.1 Introdução 19
3.2 Background 20
3.2.1 *Model checking* de um produto 20
3.2.2 Model checking de LPS 22
3.3 Exemplo 24
3.4 Tratando expressividade 25
3.4.1 Do modelo Paramétrico para o AFD 28
3.4.2 Do AFD para a Fórmula 30
3.5 Análise de Escalabilidade 31
3.5.1 Avaliação Analítica 32
3.5.2 Avaliação Prática 35
3.6 Trabalho Relacionado 37
3.7 Conclusão 37

4 **Uso de *Features* Opcionais em *Model Checking* paramétrico de LPS para análise de confiabilidade** **38**
4.1 Introdução 38
4.2 *Model Checking* paramétrico 39
4.2.1 Model checking paramétrico probabilístico 41
4.3 Abordagem Proposta 41
4.4 Prova de Validada (*Soundness*) 45
4.4.1 Demonstração 45
4.5 Conclusão 53

5 ***Model Checking* Composicional em Linhas de Produto de Software** **54**
5.1 *Model Checking* Paramétrico Composicional 55
5.2 Argumento 59
5.3 Avaliação 60
5.3.1 Análise Quantitativa 61
5.3.2 Limitações e Ameaças à Validade 66
5.4 Trabalhos Relacionados 67
5.5 Conclusão 67

6 **Conclusão** **68**
6.1 Trabalhos Futuros 69

A **Simulação Vital Signal Monitoring System** **70**
A.1 Documentação 70
A.2 Modelos 75
A.2.1 Não composicional 75
A.2.2 Composicional 78
A.3 Fórmulas 84

Referências **85**

Lista de Figuras

2.1 Visão gráfica da cadeia de Markov 8
2.2 Modelo de *features* 16
2.3 Instanciação de um produto 16

3.1 Processo de *Model Checking* 20
3.2 Exemplo de uma cadeia de Markov 21
3.3 Processo de *model-checking* paramétrico 23
3.4 Modelo de *Features* do Sistema de Monitoramento de Sinais Vitais 24
3.5 Configuração `{MONITORING,EKG,SPO2}` 25
3.6 Visão Geral do Processo de Conversão 26
3.7 AFD da Listagem 3.2 29
3.8 Extensão do modelo de *features* 35
3.9 Extensão da documentação da LPS 36
3.10 Crescimento da Fórmula com o Aumento da Variabilidade 36

4.1 Abordagem de Model Checking paramétrico para LPS 41
4.2 Avaliação da fórmula aritmética 42
4.3 Estratégia de refinamento de *features* 43
4.4 Cadeias de Markov paramétricas para LPS 43
4.5 Documentação da LPS 44
4.6 Cadeias de Markov das configurações da LPS 44
4.7 Tratamento de variabilidade 46
4.8 $G(V, E)$ 48
4.9 Tratamento de variabilidade 49

5.1 Diagramas de Sequência Detalhando uma Atividade 56
5.2 Processo de *model checking* paramétrico composicional 57
5.3 Diagrama de Atividades 58
5.4 Comparação das abordagens 60
5.5 Gráficos de Simulação 65

A.1 Modelo de Características 70
A.2 Diagrama de Atividades 71
A.3 Diagrama de sequência (Atividade: *System captures vital signal*) 71
A.4 Diagrama de sequência (Atividade: *System identifies situations*) 72
A.5 Diagrama de sequência (Atividade: *System identifies situations*) 72
A.6 Diagrama de sequência (Atividade: *System identifies situations*) 73
A.7 Diagrama de sequência (Atividade: *System identifies situations*) 73

A.8 Diagrama de sequência (Atividade: *System identifies situations*) 74
A.9 Diagrama de sequência (Atividade: *Was there any change in QoS goal?*) . 74
A.10 Diagrama de sequência (Atividade: *System configuration changes to achieve new goal*) 75

Lista de Tabelas

1.1 Capítulos e Questões de pesquisa 5

3.1 *Configuration Knowledge* 24
3.2 *Substituição de Variáveis* 29

5.1 Definição do Objetivo 61
5.2 Questões e Métricas 62
5.3 Dados Coletados 63

Capítulo 1

Introdução

Cada vez mais as pessoas dependem de softwares em seu dia a dia. Softwares são utilizados nos mais diversos campos desde telefones celulares a sistemas de controle de tráfego aéreo (Hoffman, 2008; Grunske, 2008; Lutz, 2000). É desejável que todos esses sistemas sejam confiáveis, especialmente aqueles que lidam com aspectos críticos tais como sistemas de uso médico, controle de tráfego aéreo, sistemas embarcados de segurança automotiva, entre outros.

Garantir a *dependabilidade* [1] desses sistemas, ou seja, garantir que um *software* mantenha níveis adequados de disponibilidade, confiabilidade, segurança, integridade e manutenibilidade é um problema especialmente importante, uma vez que uma falha nesses sistemas podem levar a consequências desastrosas.

Pequenos erros podem ser identificados por meio de técnicas de teste, entretanto testes não são adequados para identificar erros estruturais. Tais erros têm grande impacto no software e, portanto, devem ser identificados o quanto antes no ciclo de desenvolvimento, ainda em fase de projeto, uma vez que o custo de manutenção e evolução de *software* em fases tardias no ciclo de desenvolvimento pode ser caro ou inviável (Hoffman, 2008).

A confiabilidade, funcionamento continuamente correto do software, é uma propriedade fundamental nesse contexto (Avizienis et al., 2004). *Model-checking* é uma das técnicas utilizadas para fazer a verificação de propriedades não funcionais tais como confiabilidade (Rodrigues et al., 2012). A partir de modelos que representam a arquitetura e o comportamento do *software*, é possível estimar essas propriedades. Esses modelos podem ser obtidos a partir de artefatos de documentação do *software* tais como diagramas UML (Object Management Group, 2009).

Tais modelos possibilitam análises por meio das quais é possível identificar os componentes de maior criticidade e as práticas de *design* mais adequadas de forma aumentar sua confiabilidade (Rodrigues et al., 2012).

Ao verificar software por meio de *model checking* podemos nos deparar com o problema da explosão de estados do modelo uma vez que mesmo softwares simples podem ter milhões de possíveis estados no modelo. Assim, é preciso construir modelos em um nível de abstração cuja verificação seja viável com relação ao esforço computacional necessário (Hoffman, 2008). Projetar tais modelos envolve um esforço considerável. Mesmo

[1] Neologismo originário do termo *dependability* do inglês cuja tradução, confiabilidade, não é suficiente para expressar o conjunto mais amplo de conceitos representado pelo termo dentre os quais *reliability* é traduzido para confiabilidade.

utilizando técnicas automáticas ou semi-automáticas é necessário adequar a entrada, por exemplo diagramas, de forma a representar o software com o nível de detalhes desejado.

O desafio de garantir a confiabilidade de um *software* é ainda maior quando se trata de Linhas de Produtos de Software. Linha de produtos de software (LPS) é uma técnica de reúso estratégico que visa minimizar os custos de produção de uma família de produtos aproveitando o que há de comum e gerenciando as variabilidades de maneira sistemática (Czarnecki and Eisenecker, 2000). Uma família de produtos ou de sistemas é um conjunto de sistemas ou produtos relacionados que podem ser construídos a partir de um conjunto comum de artefatos (Clements and Northrop, 2001). Esses artefatos são agrupados em funcionalidades relevantes para algum *stakeholder* da LPS chamadas *características* (Czarnecki and Eisenecker, 2000; Jilles Van et al., 2001).

Garantir a confiabilidade de cada produto de uma LPS pode representar um grande volume de trabalho uma vez que o número de produtos cresce exponencialmente com a quantidade de *características* da LPS. A utilização de técnicas tradicionais em linhas de produtos de software (LPS) não é escalável, pois seria necessário construir um modelo e estimar suas propriedades para cada possível produto da LPS.

Cada produto de uma LPS é um software diferente. No entanto, possui diversos artefatos comuns em sua estrutura. Essas semelhanças podem ser utilizadas de forma a reduzir o esforço de verificação de produtos de uma LPS. Alguns trabalhos tratam esse problema diretamente (Classen et al., 2011, 2010; Ghezzi and Sharifloo, 2011b).

Porém, esses trabalhos impõem diferentes restrições sobre a variabilidade do *software*. Uma solução mais abrangente em termos de suporte à variabilidade ainda é um problema em aberto.

O model checking paramétrico é uma técnica que permite que a avaliação de determinadas variáveis do modelo seja adiada o máximo possível (Hahn, 2008). Por meio dessa técnica, obtém-se uma fórmula aritmética cuja valoração representa o resultado numérico da verificação realizada no modelo.

O modelo parametrizado pode ser construído de tal forma que a variabilidade da LPS seja representada por meio de parâmetros no modelo (Ghezzi and Sharifloo, 2011b). Assim, é possível verificar a confiabilidade de todos os produtos da LPS por meio de uma única verificação do modelo. O resultado da verificação é dado em termos dos parâmetros definidos no modelo por meio de uma fórmula aritmética. Essa fórmula pode então ser avaliada posteriormente com diferentes valorações resultando nos valores de confiabilidade específicos para cada produto da LPS.

1.1 Problema

Por meio do *model checking* paramétrico é possível modelar uma LPS utilizando um único modelo capaz de representar todos os seus produtos. Dessa forma, o esforço de modelagem é reduzido viabilizando uso de tal técnica em LPS. Entretanto, as soluções atuais de uso desta técnica estão limitadas a características alternativas, ou seja, aquelas que são selecionadas de maneira excludente (Ghezzi and Sharifloo, 2011b). Devido a tal restrição a técnica atual não é capaz de atender outros tipos de variabilidades comuns em uma LPS tais como opcionais e *OR*. Assim, a primeira questão de pesquisa deste trabalho é:

Questão de Pesquisa 1 É possível tratar os diferentes tipos de variabilidade de uma LPS por meio de modelo paramétrico? Se sim, como?

Por meio de uma mecanismo de desvio implementado com o uso de parâmetros, propomos uma maneira de seletivamente isolar partes do modelo permitindo a representação dos diferentes tipos de variabilidades.

Foi verificado que o mecanismo proposto pode produzir fórmulas grandes, com milhões de operandos. Fórmulas aritméticas deste tamanho podem ser avaliadas em questão de milissegundos em processadores domésticos (Intel, 2012), entretanto os parâmetros de um modelo paramétrico podem servir diferentes finalidades e, consequentemente, podem ser aplicados em diferentes contextos além da parametrização da variabilidade de uma LPS. Em particular, o cálculo de confiabilidade de uma LPS pode depender de componentes cujo valor de confiabilidade só seja conhecido em tempo de execução. Em determinados cenários, é necessário avaliar a fórmula paramétrica em tempo de execução, por exemplo, sistemas que se baseiam em decisões de tempo real podem avaliar essas fórmulas constantemente com diferentes valorações para os parâmetros. Nesse contexto o tamanho da fórmula é uma questão relevante , especialmente quando lidamos com recursos computacionais limitados, como por exemplo dispositivos móveis.

Assim, é importante saber:

Questão de Pesquisa 2 Quais fatores impactam no tamanho da fórmula parametrizada?

Conhecendo tais fatores é possível buscar estratégias que reduzam o tamanho da fórmula gerada viabilizando sua utilização em diferentes cenários, mesmo aqueles com limitada disposição de recursos computacionais. Assim, temos a seguinte questão:

Questão de Pesquisa 3 Como reduzir o tamanho da fórmula parametrizada?

1.2 Solução Proposta

Visando responder a essas questões, foi proposta uma estratégia de modelagem de variabilidades para cálculo de confiabilidade de LPS utilizando técnica de *model checking* paramétrico. A técnica proposta trata variabilidade por meio de mecanismo de desvio cujo princípio de funcionamento foi demonstrado por meio de teoria dos grafos. O mecanismo de desvio é capaz de tratar variabilidades opcionais por meio de probabilidades parametrizadas no modelo. Por ser capaz de tratar variabilidades do tipo opcional, a técnica proposta é capaz de tratar qualquer tipo de variabilidade (Questão 1).

O objetivo da verificação é especificado por meio de expressões de lógica temporal. Este trabalho foca no uso de expressões que tem por objetivo especificar a probabilidade de funcionamento correto do sistema ao longo de um tempo ilimitado. A lógica temporal pode ser aplicada para outros propósitos como por exemplo, saber quais configurações

satisfazem um determinado inferior de confiabilidade, entretanto, esse trabalho estuda apenas expressões para cálculo de confiabilidade em tempo ilimitado.

PARAM é ferramenta utilizada para verificar o modelo construído por meio da abordagem proposta e gerar a fórmula paramétrica (Hahn et al., 2010). Foi conduzido um estudo a fim de verificar que aspectos da modelagem impactam no crescimento da fórmula obtida (Questão 2).

A partir desse estudo propomos uma extensão por meio da qual é possível reduzir o tamanho da fórmula final obtida. A extensão proposta divide um modelo único maior em modelos menores utilizados para gerar partes da fórmula que são posteriormente recombinadas. Após essa recombinação é obtida uma fatoração parcial da fórmula paramétrica que pode levar a uma quantidade menor de operandos por meio da eliminação de redundâncias (Questão 3). Foi observado que a redução da redundância de parâmetros por si só não é suficiente para reduzir a fórmula em qualquer caso entretanto a abordagem proposta permite ao engenheiro de aplicação a utilização da fórmula fatorada ou da fórmula completamente expandida caso deseje.

De acordo com a taxonomia proposta por von Rhein et al. (2013), a abordagem proposta é categorizada para cada uma das três dimensões propostas (Amostragem, Agrupamento, Codificação da Variabilidade) da seguinte forma: a amostragem considera todos os produtos válidos, o agrupamento é por produto (não por *feature*) e a codificação da variabilidade é baseada na família como um todo (e não produto a produto).

1.3 Contribuições

Precisamente, as contribuições desse trabalho são as seguintes:

- Método escalável de verificação de confiabilidade em LPS para qualquer tipo de variabilidade.
- Formalização do mecanismo de tratamento de variabilidade do método proposto.
- Estudo da composicionalidade de modelos paramétricos como forma de reduzir o tamanho da fórmula.

O seguinte artigo foi resultado deste trabalho:

- Vinícius Nunes, Paula Fernandes, Vander Alves e Genaína Rodrigues. Variability Management of Reliability Models in Software Product Lines: an Expressiveness and Scalability Analysis. In *Brazilian Symposium on Software Components, Architectures and Reuse*, SBCARS, 2012. (Publicado)

1.4 Organização do Trabalho

O Capítulo 2 apresenta os principais conceitos relacionados a *model-checking* e LPS utilizados nesse trabalho. O conteúdo do trabalho compreende um artigo listado na Seção 1.3 produzidos no contexto de pesquisa do problema estudado e de um dois capítulos adicionais onde apresentamos a formalização e euma extensão à proposta apresentada.

Nos dois primeiros é apresentada a proposta de modelagem para LPS (Capítulo 3) e sua formalização (Capítulo 4); no Capítulo 5 o conceito de composicionalidade é explorado na construção e verificação dos modelos. Por fim, o Capítulo 6 apresenta a conclusão final do trabalho.

Note que cada capítulo aborda parte do problema de maneira auto-contida. De forma a manter a homogeneidade na apresentação do conteúdo. Devido a isso, cada capítulo traz sua própria conclusão e seção de trabalhos relacionados. Além disso, é possível notar um grau de sobreposição em relação os problemas abordados nesses capítulos. A Tabela 1.1 apresenta um mapeamento entre as questões de pesquisa apresentadas na Seção 1.1 e os capítulos deste trabalho.

Capítulo	Questões de Pesquisa
3	1, 2
4	1
5	3

Tabela 1.1: Capítulos e Questões de pesquisa

Capítulo 2

Fundamentação Teórica

Nesta seção, serão apresentados os principais conceitos utilizados neste trabalho. Inicialmente serão apresentados os conceitos relacionados a dependabilidade, em seguida *model checking*, e por fim conceitos relacionados a linha de produtos de software.

2.1 Análise de Dependabilidade

Dependabilidade de um sistema é a habilidade de evitar que serviços falhem mais frequente e severamente que o aceitável. Esse conceito é formado pelos seguintes atributos (Avizienis et al., 2004):

- Disponibilidade: prontidão para execução correta.
- Confiabilidade: continuidade da execução correta.
- Segurança (*safety*): a execução do sistema não tem consequências catastróficas para o usuário ou para o ambiente.
- Integridade: a execução do sistema não faz alterações impróprias no mesmo.
- Manutenibilidade: facilidade para de se modificar ou reparar o sistema.

A análise de dependabilidade de um sistema é o estudo dessas propriedades e esta pode ser feita por meio de *model-checking* (Rodrigues et al., 2012). Dessa forma, o sistema e seus componentes são modelados e suas propriedades de interesse avaliadas. As técnicas de modelagem e análise propostas ao longo desse trabalho se restringem à verificação da propriedade de *confiabilidade*.

Essa análise é feita por meio de cadeias de Markov, modelo probabilístico definido por um conjunto de estados e um conjunto de transições entre estados com probabilidades associadas (Kay, 2006; Rodrigues et al., 2012). Essas probabilidades independem dos estados anteriores, assim, o sistema pode ser modelado por meio de uma cadeia de Markov com probabilidades associadas representando a chance de execução correta de cada componente. A valor confiabilidade da execução dos componentes é estimado por especialistas de domínio; assim, a confiabilidade global do sistema, derivada da confiabilidade dos componentes, pode ser calculada por meio da verificação da probabilidade de se chegar a um estado final, ou seja, a execução correta de todos os componentes do sistema até o final do cenário de execução avaliado.

Existem ferramentas capazes para auxiliar a criação e execução desses modelos, duas delas em particular são relevantes para este trabalho: a ferramentas PRISM (Kwiatkowska et al., 2011), que permite a modelagem, simulação e verificação de propriedades do modelo probabilístico e a ferramenta PARAM (Hahn et al., 2010) que permite a verificação de propriedades de modelos probabilísticos parametrizados. Dentre as diversas propriedades, a probabilidade de se alcançar um determinado estado em algum momento (*reachability*) é uma propriedade importante na análise de dependabilidade para avaliar a confiabilidade do sistema (Rodrigues et al., 2012).

2.2 *Model Checking* Probabilístico

Este trabalho utiliza a técnica de *model checking* probabilístico para verificar modelos. Essa técnica utiliza modelos de estados e transições onde cada transição ocorre com uma determinada probabilidade. As propriedades são verificadas por meio da propriedade de *rechability* desses modelos. Essa propriedade permite verificar a probabilidade do modelo alcançar a partir de um determinado estado um outro estado em número limitado ou ilimitado de passos.

Para fazer essas verificações foram utilizadas duas ferramentas: PRISM e PARAM. A primeira foi utilizada em etapas investigativas do trabalho, a segunda, mais amplamente explorada é capaz de descrever as propriedades verificadas em termos de parâmetros, característica amplamente explorada neste trabalho.

2.2.1 Cadeias de Markov

Uma cadeia de Markov é composta por estados e transições. Essa cadeia é utilizada para representar a dependência entre os experimentos. Dessa forma a cadeia se inicia em um determinado estado e transita de um estado para outro de acordo com a probabilidade das transições. A transição de um estado a outro é chamado de *passo*. A cada passo a probabilidade de se alcançar um próximo estado independe das probabilidades dos estados anteriores (Grinstead and Snell, 2006; Kay, 2006).

A cadeia de Markov pode ser representada por uma matriz n por n onde n é o número de estados da cadeia e cada posição ij da matriz representa a probabilidade de se transitar do estado i para o estado j. Veja o exemplo abaixo:

$$M = \begin{array}{c} \\ s0 \\ s1 \\ s2 \\ s3 \\ s4 \\ s5 \end{array} \begin{array}{c} \begin{array}{cccccc} s0 & s1 & s2 & s3 & s4 & s5 \end{array} \\ \begin{pmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Repare que todas as linhas da matriz somam 1 no total. Ou seja, para cada estado i existem uma ou mais transições de se alcançar o próximo estado de tal forma que a chance de ocorrer alguma transição é de 100% a cada passo.

Esses estados e transições podem ser representados também na forma de um grafo direcionado cujas transições são rotuladas pelas probabilidades das mesmas ocorrerem. Veja a Figura 2.1.

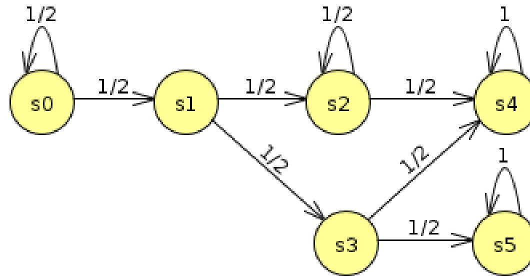


Figura 2.1: Visão gráfica da cadeia de Markov

Uma *execução* de uma cadeia de Markov é o cálculo da probabilidade de se alcançar um estado j a partir de um estado i em um determinado tempo, seja esse finito ou infinito. Na seção 2.2.2 será apresentado como essas execuções podem ser especificadas em lógica temporal por meio de uma expressão PCTL.

O cômputo da *execução* de um DTMC é feito a partir dos caminhos no grafo que representa o DTMC. Um caminho é uma sequência não vazia composta por estados do DTMC. Por exemplo, considere a cadeia de Markov do grafo 2.1, um possível caminho P entre $s0$ e $s5$ pode ser denotado por:

$P = \{s0, s1, s3, s5\}$

Outro caminho possível:

$P = \{s0, s0, s1, s3, s5\}$

Observe que os caminhos podem ter repetições dos estados. Dessa forma, entre s0 e s5 há infinitos caminhos onde o estado s0 ocorre ao menos uma vez. Ou seja, todos os possíveis caminhos serão da seguinte forma:

$P = \{s0_1, ..., s0_n, s1, s3, s5\}$
onde $1 \leq n \leq \infty$

O subconjunto dos possíveis caminhos considerados para calcular uma *execução* é determinado pelo limite de tempo definido para a execução.

Nesse contexto, o tempo discreto do DTMC é determinado pelo número de *passos* de cada caminho. Conforme mencionado anteriormente, um *passo* é uma transição entre estados. Dessa forma cada um dos caminhos exemplificados acima possui um número de passos $n = |P| - 1$, onde $|P|$ é o tamanho da sequência de estados P que compõem cada caminho.

Para computar uma execução é necessário calcular todos os possíveis caminhos entre os estados inicial e final que satisfaçam as condições impostas, em particular as expressões relativas a tempo. Por exemplo:

(1) Qual a probabilidade de se alcançar o estado $s5$ a partir de $s0$ em no máximo 10 passos?

A sentença acima aplicada ao modelo da Figura 2.1 limita os possíveis caminhos entre $s0$ e $s5$ aos seguintes:

1. {s0,s1,s3,s5}
2. {s0,s0,s1,s3,s5}
3. {s0,s0,s0,s1,s3,s5}
4. {s0,s0,s0,s0,s1,s3,s5}
5. {s0,s0,s0,s0,s0,s1,s3,s5}
6. {s0,s0,s0,s0,s0,s0,s1,s3,s5}
7. {s0,s0,s0,s0,s0,s0,s0,s1,s3,s5}
8. {s0,s0,s0,s0,s0,s0,s0,s0,s1,s3,s5}

Como cada estado $s0$ é considerado, probabilisticamente, um evento independente, a probabilidade de se alcançar o estado final de cada caminho é dada pela multiplicação desses eventos independentes. Ou seja a probabilidade de se alcançar o estado $s5$ por meio do caminho do Item 1 é dada por:

$0.5 * 0.5 * 0.5 = 0.125$

A probabilidade de se alcançar o estado $s5$ por meio dos demais caminhos é obtida de forma análoga. Assim temos, para cada caminho enumerado acima, os seguintes valores:

1. $0.5 * 0.5 * 0.5 = 0.125$
2. $0.5 * 0.5 * 0.5 * 0.5 = 0.0625$
3. $0.5 * 0.5 * 0.5 * 0.5 * 0.5 = 0.03125$
4. $0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 = 0.015625$
5. $0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 = 0.0078125$
6. $0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 = 0.00390625$
7. $0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 = 0.001953125$
8. $0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5 = 0.0009765625$

De posse das probabilidades de se alcançar o estado $s5$ a partir de todos os caminhos, estas devem ser somadas para obter a probabilidade final de acordo com a expressão exemplo utilizada nesse exemplo.

Somando-se todos esses valores obtemos uma probabilidade final de : 0.2490234375 ou de 24,0234375%.

Considere agora o seguinte exemplo:

(2) Qual a probabilidade de se alcançar o estado $s5$ *a partir de* $s0$ *em uma quantidade qualquer de passos?*

A sentença acima aplicada ao modelo da Figura 2.1 determina um número infinito de caminhos entre $s0$ e $s5$:

1. {s0,s1,s3,s5}
2. {s0,s0,s1,s3,s5}
3. {s0,s0,s0,s1,s3,s5}
   .
   .
   .
4. $\{s0_1, ..., s0_n, s1, s3, s5\}$
   onde $1 \leq n \leq \infty$

Observe que o passo $s0$ para $s0$ ocorre zero vezes no caminho do Item 1 e infinitas vezes no caminho denotado de forma abstrata no Item 4.

Seja s_{ij} o passo que representa a trasição entre si e sj cuja probabilidade está expressa na matriz utilizada para obter o grafo dos exemplos nas linhas e colunas indexadas por i e j respectivamente.

Assim, podemos reescrever cada um dos caminhos em termos de suas transições:

1. $\{s_{01}, s_{13}, s_{35}\}$
2. $\{s_{00}, s_{01}, s_{13}, s_{35}\}$
3. $\{s_{00}, s_{00}, s_{01}, s_{13}, s_{35}\}$
   .
   .
   .
4. $\{s_{00}, ..., s_{00}, s_{01}, s_{13}, s_{35}\}$
   onde s_{00} ocorre zero ou mais vezes

O valor de probabilidade de cada caminho pode então ser expresso em termos dos passos s_{ij} que compõem cada caminho.

1. $s_{01} * s_{13} * s_{35}$
2. $s_{00} * s_{01} * s_{13} * s_{35}$
3. $s_{00} * s_{00} * s_{01} * s_{13} * s_{35}$
   .
   .
   .
4. ${s_{00}}^n * s_{01} * s_{13} * s_{35}$

Somando-se todas essas expressões e colocando em evidência o termo s_{00} obtemos o seguinte:
$(\sum_{n=0}^{\infty} {s_{00}}^n) * s_{01} * s_{13} * s_{35}$
Observe que s_{00} sempre assumirá valores entre 0 e 1 e considere os seguintes casos:

- Caso seja 0, o resultado será indefinido uma vez que as transições partindo de $s0$ somarão menos que 1.
- Caso seja 1, o resultado do somatório será infinito.
- Caso contrário $0 < s_{00} < 1$ e o somatório pode ser considerado uma soma infinita de uma série geométrica, cujo valor total da soma é dado por:

$$\frac{1}{1 - s_{00}}$$

Este último é o caso do nosso exemplo, assim a probabilidade de se alcançar o estado $s5$ a partir do estado $s0$ em uma quantidade qualquer de passos é dado por:
$(\frac{1}{1-s_{00}}) * s_{01} * s_{13} * s_{35}$
Substituindo-se os valores de probabilidades de cada transição de acordo com a matriz em questão obtemos o seguinte:
$(\frac{1}{1-0.5}) * 0.5 * 0.5 * 0.5) = 0.25$
ou 25%
Os exemplos apresentados nessa seção serão revisitados nas seções seguintes pois são fundamentais para o entendimento do trabalho.

2.2.2 Lógica Temporal

Nesta seção serão apresentados os principais conceitos relacionados à construção de expressões PCTL (*Probabilistic Computation Tree Logic*) (Hansson and Jonsson, 1994) para o PRISM, em particular as expressões que permitem especificar a busca da probabilidade de se chegar a um determinado estado seja em tempo limitado ou não uma vez que estas são as expressões utilizadas no restante do trabalho.

Uma expressão PCTL pode ser uma fórmula de estados ou de caminhos. A primeira descreve propriedades dos estados a serem avaliadas em um sistema de transições, em particular uma cadeia de Markov, a segunda descreve propriedades a serem observadas em um caminho da cadeia de Markov. Essas expressões são definidas indutivamente da seguinte maneira (Hansson and Jonsson, 1994; Kwiatkowska et al., 2007):

1. Cada proposição atômica é uma fórmula de estados.
2. Sejam $f1$ e $f2$ fórmulas de estados, então as proposições compostas utilizando os operadores $\wedge$, $\vee$, $\neg$ e $\rightarrow$ também o são. Ex $(f1 \vee f2)$.
3. Sejam $f1$ e $f2$ fórmulas de estados e t um inteiro não negativo ou ∞, então $f1U^{\leq t}f2$ e $f1W^{\leq t}f2$ são fórmulas de caminhos.
4. Seja f uma fórmula de caminho e p um número real tal que $0 \leq p \leq 1$, então $[f]_{\sim p}$ é uma fórmula de estado.

Onde $\sim$ é qualquer operador do conjunto $\{>, <, \leq, \geq\}$ e p representa um valor de probabilidade e t é o tempo representado pelo número de *passos* na cadeia de Markov.

O operador U *(until)* é o operador "até que". Para que um caminho atenda uma expressão $f1U^{\leq t}f2$, a proposição f1 deve ser verdadeira até que f2 se torne verdadeira, ou seja, em todos os estados anteriores ao estado que torna f2 verdadeira f1 deve ser verdadeira.

O operador W *(Weak until or unless)* é o operador "a menos que". Para que um caminho atenda uma expressão $f1W^{\leq t}f2$, a proposição f1 deve ser verdadeira a menos que f2 se torne verdadeira, ou seja f2 não necessariamente precisa ser verdadeira, basta que f1 seja sempre verdadeira.

Adicionalmente o operador P é utilizado para verificar a probabilidade de uma expressão ser verdadeira (Kwiatkowska et al., 2011) e possui o seguinte formato:

Seja f uma fórmula de caminho e p um número real tal que $0 \leq p \leq 1$, então:

$P \sim p[f]$
Onde $\sim$ é qualquer operador do conjunto $\{>, <, \leq, \geq\}$ e p representa um valor de probabilidade.

Esse operador é verdadeiro se a probabilidade de que os caminhos da cadeia satisfaçam a fórmula de caminho f seja $\sim p$ (Ex: ≤ 0.87). Esse operador também permite operações quantitativas e não apenas verdadeiro ou falso. Essa característica é importante para o trabalho apresentado nos capítulos subsequentes. Esse operador pode assumir o seguinte formato:

Seja f uma fórmula de caminho, então: $P =?[f]$

Esse operador retorna um valor p real tal que $0 \leq p \leq 1$, representando a probabilidade de que a fórmula de caminho f seja verdadeira.

Exemplos de PCTL:
Considere a expressão (1) do exemplo anterior:
(1) Qual a probabilidade de se alcançar o estado $s5$ *a partir de* $s0$ *em no máximo 10 passos?*
A expressão PCTL equivalente ao enunciado seria:

$P =?[trueU^{\leq 10}s5]$

Considere agora a expressão (2) do exemplo anterior
(2) Qual a probabilidade de se alcançar o estado $s5$ *a partir de* $s0$ *em uma quantidade qualquer de passos?*

A expressão PCTL equivalente ao enunciado seria:
$P =?[trueU^{\leq \infty}s5]$

Observe que em ambos os casos $s0$ não faz parte da expressão uma vez que o mesmo é utilizado apenas para identificar o ponto de partida na cadeia.

Neste trabalho usaremos predominantemente expressões PCLT que especificam *reachability* em tempo ilimitado, em particular, expressões de existência probabilística de tempo ilimitado (Grunske, 2008).

2.2.3 PRISM

O PRISM é uma ferramenta de model-checking probabilístico. Essa ferramenta pode ser utilizada para fazer análises formais do comportamento aleatório ou probabilístico de sistemas (Kwiatkowska et al., 2011).

Por meio de modelos que representam a arquitetura de um sistema, seus componentes e suas interações a ferramenta PRISM é capaz de verificar, por meio de diferentes técnicas, a probabilidade de se alcançar determinado estado do sistema (*reachability*).

A ferramenta PRISM suporta diferentes tipos de modelos probabilísticos, em particular o modelo DTMC (*discrete-time Markov chains*) permite que as transições do modelo sejam feitas por meio de escolhas probabilísticas.

Neste trabalho utilizaremos apenas modelo do tipo DTMC ou cadeias de Markov de tempo discreto. A escolha dessa técnica se deve ao fato de podermos modelar as diferentes transições de um sistema em modelos que considerem chance aleatória de falha dos componentes e podermos identificar os diferentes estados de execução dos componentes do sistema de maneira discreta.

Essa técnica é utilizada para calcular a propriedade de *reachability*. Por meio dessa propriedade é possível saber qual a probabilidade do sistema representado no modelo alcançar determinado ponto em sua execução. Diferentes semânticas podem ser atribuídas à probabilidade de se alcançar um determinado estado, como por exemplo, a confiabilidade (*reliability*) da aplicação.

2.2.4 Linguagem PRISM

Esta seção apresenta os conceitos da linguagem de modelagem da ferramenta PRISM restritos àqueles são utilizados nos exemplos ao longo desse trabalho.

A linguagem PRISM é uma linguagem de modelagem baseada em estados derivada do formalismo de módulos reativos (Alur and Henzinger, 1996). Por meio dessa linguagem de mais alto nível é possível especificar modelos que representam cadeias de Markov a partir da quais propriedades podem ser verificadas. Essas propriedades são expressas por meio de expressões PCTL (Baier and Katoen, 2008; Hansson and Jonsson, 1994; Bianco and Alfaro, 1995). Um modelo especificado nessa linguagem é chamado de modelo PRISM.

Tais modelos contém um ou mais módulos especificados por meio da palavra reservada `module`. Cada módulo representa um processo independente que executa em paralelo com os demais módulos. Nesses modelos é possível declarar variáveis e constantes. As variáveis são definidas em um intervalo de valores inteiros e precisam especificar um valor inicial dentro desse intervalo. As variáveis podem ser declaradas nos escopos global e de módulo, as constantes são declaradas no escopo global. As variáveis declaradas no escopo global podem ser lidas e alteradas por qualquer módulo, as variáveis no escopo de módulo podem ser lidas por qualquer módulo, porém só podem ser alteradas pelo módulo que a declara.

A cadeia de Markov é sintetizada a partir da composição paralela de todos os módulos (Alur and Henzinger, 1996). Cada estado dessa cadeia é determinado por um estado de valoração do conjunto de variáveis globais e de módulo do modelo.

Cada módulo é composto por uma série de comandos. Cada comando define um estado e as transições que partem desse estado. A Listagem 2.1 apresenta um exemplo de comando:

Listagem 2.1: Comando PRISM

```
[ação] <condição> -> <expressão> : <atualização de variáveis>;
```

Uma *condição* é um predicado definido sobre qualquer variável ou constante do modelo. Uma vez satisfeito o predicado, o modelo *atualizará um conjunto de variáveis* de acordo com a probabilidade especificada por uma *expressão* (que pode ser uma constante). Cada expressão pode envolver várias constantes reais resultando em número real p tal que $0 \leq p \leq 1$, que representa a probabilidade do modelo fazer aquela transição. Cada atualização de variáveis representa uma transição de estado modelo como um todo. Um comando pode apresentar diversos pares de `<expressão> : <atualização de variáveis>`, separados pelo operador $'+'$ representando várias transições a partir do estado representado pelo mesmo.

Ações são utilizadas para sincronizar transições em módulos distintos e são especificadas por meio de identificadores textuais entre colchetes declaradas no início de um comando (ver Listagem 2.1). Na cadeia de Markov sintetizada o modelo fará uma única transição para o estado resultante de todas as atualização de variáveis de todos os comandos sincronizados sob uma mesma ação. Caso não exista nenhum outro comando com a mesma ação esta não tem efeito sob o modelo. Caso um módulo alcance um estado de sincronização antes dos demais este ficará bloqueado até que os demais cheguem a seus comandos sob a mesma ação.

2.2.5 PARAM

PARAM é uma ferramenta de *model checking* probabilístico paramétrico (Hahn et al., 2010). Seu princípio de funcionamento é o mesmo da ferramenta PRISM, porém, esta é capaz de gerar uma fórmula parametrizada por parâmetros definidos no modelo cuja valoração resulta em valores de probabilidade para a propriedade verificada no modelo.

O PARAM utiliza como linguagem de modelagem uma extensão da linguagem PRISM. Essa extensão define a palavra reservada `param`, utilizada para definir parâmetros no modelo. Esses parâmetros podem ser utilizados para parametrizar valores de probabilidades no modelo que irão parametrizar o resultado final. Nessa ferramenta o resultado final não é apenas um número, mas uma fórmula.

O PARAM utiliza a mesma sintaxe para especificar expressões em lógica temporal que são utilizadas para verificar as propriedades do modelo.

Modelos PARAM

A ferramenta PARAM utiliza a mesma linguagem da ferramenta PRISM com adição da palavra reservada `param` que permite declarar o equivalente a constates da linguagem PRISM, porém não valoradas. Ao longo do trabalho os termos *módulo PARAM*, *estados*

do PARAM e *transições do PARAM* são utilizados de maneira intercambiável com os termos *módulo PRISM*, *estados do PRISM* e *transições do PRISM* respectivamente.

2.3 Linha de Produtos de Software

Linha de produtos de software (LPS) é uma técnica de reúso que visa minimizar os custos de produção de uma família de produtos aproveitando o que há de comum e gerenciando as variabilidades. Uma família de produtos ou de sistemas é um conjunto de sistemas ou produtos relacionados que podem ser construídos a partir de um conjunto comum de artefatos de forma sistemática (Clements and Northrop, 2001). Assim, uma LPS é formada por diversos produtos de uma mesma família, a produção de um produto específico por meio da LPS é chamado instanciação do produto (Jilles Van et al., 2001). A LPS possui um conjunto de artefatos comuns chamado base de artefatos. Por meio da composição desses artefatos é possível instanciar cada um dos produtos da LPS; dessa forma, cada produto da LPS é formado por um subconjunto selecionado dos artefatos da LPS.

Variabilidade é a habilidade de mudar ou customizar um sistema (Jilles Van et al., 2001). Em uma LPS um ponto de variabilidade é um ponto de diferenciação entre produtos. Para gerenciar variabilidade é preciso que estas sejam restringidas, ou seja, as possíveis variantes de cada ponto de variabilidade precisam estar especificadas e representadas formalmente (Krueger, 2003).

Cada ponto de variabilidade estabelece restrições com relação a quais *features* da LPS podem ser consideradas ou não para instanciar o produto em questão. Uma *feature* é um aspecto do sistema importante para algum *stakeholder* (Czarnecki and Eisenecker, 2000). As *features* e as variabilidades podem ser representadas por meio de um modelo de *features*. Esse modelo hierárquico mapeia as *features* aos pontos de variabilidades da LPS, restringindo a maneira como essas podem ser combinadas a fim de delimitar de maneira consistente quais *features* da linha de produtos devem ser selecionadas para se obter uma instância da LPS. As principais relações entre as *features* em um determinado ponto de variabilidade em um modelo de *features* são (Czarnecki and Eisenecker, 2000):

- Obrigatórias: todo produto apresentará essa *features*
- Alternativas: um produto pode apenas selecionar uma dentre as várias *features* alternativas
- Opcionais: um produto pode ou não apresentar essa *feature*.
- OR: um produto pode ter uma ou mais *features* de um conjunto de *features* relacionadas.

A Figura 2.2 apresenta um modelo de *features* com exemplos de *features* obrigatórias, opcionais e alternativas e OR. Além dessas restrições, é possível estabelecer restrições por meio de expressões de lógica proposicional denominadas *cross-tree-constraints*. As expressões lógicas apresentadas na parte inferior da Figura 2.2 representam *cross-tree-constraints*. Cada *feature* selecionada é avaliada como verdadeiro e cada *feature* não selecionada é avaliada como falsa.

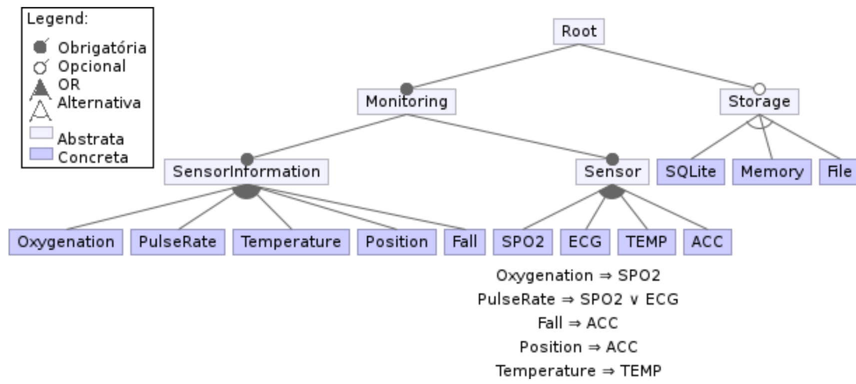


Figura 2.2: Modelo de *features*

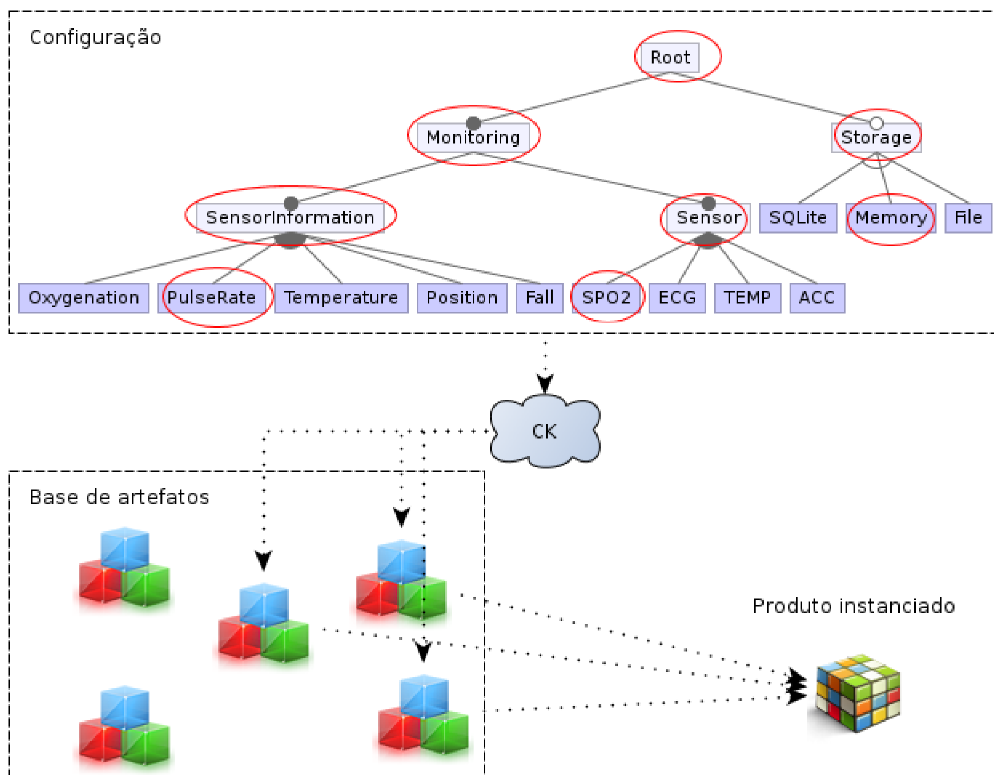


Figura 2.3: Instanciação de um produto

Uma particular resolução das variabilidades expressas no modelo *features* é chamada configuração. Cada configuração determina um produto da LPS. Uma configuração que respeite as restrições impostas sob a seleção das *features* é dita uma configuração *válida*.

Além da seleção das *features* em cada ponto variabilidade é necessário saber quais artefatos da base de artefatos são utilizados de acordo com cada *feature* selecionada para pode instanciar o produto. A LPS possui um mapeamento entre artefatos e *features* chamado *configuration knowledge* (CK) (Czarnecki and Eisenecker, 2000).

As *features* podem ser ainda classificadas em abstratas e concretas (Thüm et al., 2011). Características abstratas são *features* utilizadas para estruturar o modelo de *features* ou

agrupar um conjunto de *features* porém não possuem artefatos associados a elas. As *features* ditas concretas são aquelas que possuem artefatos associados. A Figura 2.2 apresenta exemplos de *features* abstratas e concretas.

Assim, para instanciar um produto em particular da LPS, é necessário o modelo de *features*, uma configuração, o CK e a base de artefatos. Dessa forma, é possível obter o mapeamento desde as *features* desejadas pelo usuário até o seu produto final. A Figura 2.3 ilustra esse processo. As *features* circuladas com a cor vermelha representam as *features* selecionadas.

2.3.1 Número de configurações

O número de possíveis configurações de uma LPS depende diretamente das restrições estabelecidas para a seleção das *features*. Considerando o pior caso, um modelo de *features* onde todas as *features* são opcionais, a LPS disporia de $(2^n - 1)$ configurações diferentes (note que uma configuração onde nenhuma *feature* é selecionada não é considerada uma configuração válida).

Devido a isso, os problemas que lidam com LPS precisam endereçar o problema do número exponencial de configurações para que possam assegurar soluções escaláveis.

Capítulo 3

Gerência de Variabilidade de Modelos de Confiabilidade em Linhas de Produtos de Software: uma Análise de Escalabilidade e Expressividade

Em alguns domínios, especialmente os de sistemas críticos, exigem softwares dependáveis [1]. Garantir dependabilidade não é um problema trivial. *Model-checking* pode ser utilizado para estimar confiabilidade de software por meio de modelos que representam o comportamento do sistema. Através desses modelos é possível estimar e medir quantitativamente propriedades tais como confiabilidade. No contexto de Linhas de Produtos de Software (LPS), precisamos verificar uma família de sistemas. Não é viável construir um modelo para cada configuração da LPS uma vez que o número de modelos necessário pode ser grande. Algumas contribuições tratam diretamente esse problema propondo técnicas especificamente desenvolvidas para LPS. Em particular, a técnica de *model-checking* paramétrica permite o uso de um único modelo para obter valores de propriedades de diferentes configurações a partir de uma fórmula aritmética. Entretanto, mesmo uma fórmula aritmética pode não ser fácil de avaliar. Se o número de operandos for grande o suficiente o custo de avaliação da fórmula pode também ser alto. Técnicas atuais impõem limitações sobre a variabilidade e/ou a arquitetura do sistema. Até onde sabemos, tratar variabilidade em *model-checking* ainda é um problema em aberto. Este trabalho é uma investigação de todo o processo de obtenção da fórmula paramétrica aritmética para uma LPS. Conhecendo esse processo e os fatores que afetam diretamente o crescimento da fórmula, é possível desenvolver novas técnicas para lidar com *model-checking* paramétrico em LPS com menos restrições.

[1] Neologismo originário do termo *dependable* do inglês cuja tradução, confiabilidade, no é suficiente para expressar o conjunto mais amplo de conceitos representado pelo termo dentre os quais *reliability* é traduzido para confiabilidade.

3.1 Introdução

Garantir a dependabilidade de um software , ou seja, garantir que um software tem níveis adequados de disponibilidade, confiabilidade, segurança (*security*), integridade e manutenibilidade é um problema especialmente importante para sistemas críticos, uma vez que uma falha nesses sistemas pode levar a consequências desastrosas.

Em particular, a confiabilidade, continuidade da correta operação do software, é uma propriedade fundamental nesse contexto (Avizienis et al., 2004). *Model-checking* é uma técnica que pode ser utilizada para verificar propriedades não funcionais tais como confiabilidade. Utilizando artefatos de documentação do software como entrada, por exemplo diagramas UML, é possível construir modelos a partir dos quais a confiabilidade do software é estimada (Rodrigues et al., 2012).

A dependabilidade de um software deve ser avaliada tão logo quanto possível no ciclo de desenvolvimento de software, preferencialmente na fase de projeto, uma vez que os custos de manutenção e evolução de um software em etapas posteriores pode ser cara ou inviável (Hoffman, 2008). Através dessa análise nós podemos identificar os componentes mais críticos e as práticas de arquitetura mais apropriadas de forma a mitigar a chance de falha do software e, dessa forma, aumentar sua confiabilidade (Rodrigues et al., 2012).

O problema é ainda mais difícil ao lidar com Linhas de Produtos de Software (LPS) (Clements and Northrop, 2001). Numa LPS cada produto é um software diferente apesar de possuir artefatos comuns em sua estrutura. Estimar a confiabilidade de cada produto utilizando técnicas tradicionais em cada produto separadamente pode levar a um grande volume de trabalho uma vez que o número de produtos cresce exponencialmente com o número de *features* de uma LPS e seria necessário construir um modelo de confiabilidade para cada um destes.

Alguns trabalhos tratam diretamente esse problema (Classen et al., 2011, 2010; Ghezzi and Sharifloo, 2011b). A estratégia desses trabalhos consiste em construir um único modelo representando todos os produtos da LPS. Isso pode ser feito utilizando *model-checking* paramétrico (Hahn, 2008). Através dessa técnica, é possível obter uma fórmula aritmética cuja avaliação represente um valor numérico da propriedade verificada no modelo. A parametrização permite a representação da variabilidade da LPS em um único modelo, através de diferentes atribuições de valores para os parâmetros é possível representar diferentes produtos (Ghezzi and Sharifloo, 2011b).

Entretanto, as abordagens atuais impõe restrições sobre a expressividade da LPS, ou seja, restrições sobre sua variabilidade e/ou restrições sobre sua arquitetura. Essas restrições vão desde de premissas sobre o mapeamento entre *features* e artefatos até limitações sobre a variabilidade tais como tratar apenas *features* `Alternativas`. Assim, esse problema carece de uma abordagem mais escalável e abrangente.

Diferentes estratégias podem ser utilizadas para modelar variabilidade e essas estratégias afetam diretamente o tamanho final da fórmula. Esse tamanho deve ser limitado de tal forma que sua avaliação seja viável uma vez que a explosão no crescimento do número de operandos da fórmula pode tornar sua avaliação impraticável sob determinadas condições.

Este trabalho apresenta um estudo analítico do processo de conversão de um modelo paramétrico para uma fórmula aritmética e uma abordagem para lidar com o problema de expressividade enfatizando decisões que impactam o tamanho final da fórmula e, con-

sequentemente, o custo de avaliação. Através desse estudo é possível desenvolver novas técnicas paramétricas capazes de lidar com variabilidade eficientemente e com menos restrições à expressividade. As principais contribuições desse trabalho são:

- **Expressividade:** Este trabalho apresenta uma estratégia para aprimorar a expressividade e como esta pode ser utilizada para tratar *features* opcionais.
- **Análise de escalabilidade:** Este trabalho apresenta uma análise completa do processo de *model-checking* paramétrico aplicado a uma LPS. É discutido o tamanho da fórmula e as implicações práticas de avaliar fórmulas grandes.

A Seção 4.2 detalha o problema e introduz alguns conceitos de *model-checking* necessários para um melhor entendimento das seções seguintes. A Seção 3.3 apresenta o exemplo que será utilizado ao longo do trabalho, a Seção 3.4 apresenta a abordagem de modelagem e detalha o *model-checking* paramétrico, a Seção 3.5 destaca os principais aspectos que impactam no tamanho da fórmula a partir de perspectivas práticas e analísticas e mostra como estender as abordagens existentes para equilibrar escalabilidade e expressividade. A Seção 3.6 discute os trabalhos relacionados apresentados ao longo da análise. Por fim, a Seção 4.5 apresenta a conclusão.

3.2 Background

Avaliar a dependabilidade de software é um problema importante, especialmente quando se trata de sistemas críticos. Estimar a confiabilidade do software nas fases iniciais do ciclo de desenvolvimento permite que decisões importantes sejam tomadas ainda na fase de projeto. Por meio de uma análise de sensibilidade dos componentes, é possível identificar quais componentes são mais críticos do software quantitativamente.

Essa seção apresenta os passos para *model-checking* de um produto e para uma LPS introduzindo conceitos e ferramentas relacionados.

3.2.1 *Model checking* de um produto

Model checking pode ser feito antes do desenvolvimento utilizando modelos comportamentais para construir um modelo que represente o software (passo 1).

Esses modelos, utilizados como entrada para ferramentas de *model checking*, permitem a verificação de propriedades tais como confiabilidade (passo 2). Fig. 3.1 apresenta os passos desse processo.

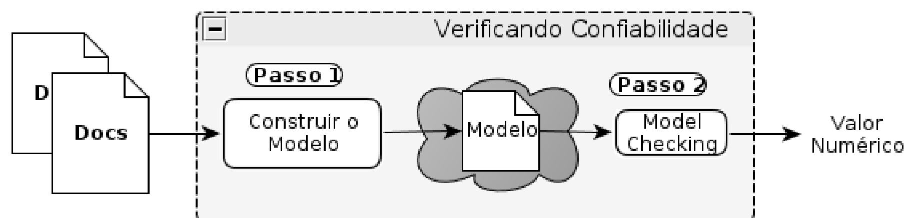


Figura 3.1: Processo de *Model Checking*

Model checking pode ser feito por meio de ferramentas de *model checking* probabilístico como o PRISM (Passo 2 da Fig 3.1). A ferramenta PRISM utiliza cadeias de Markov para verificar propriedades tais como confiabilidade em um model.

Cadeias de Markov desempenha um papel fundamental nesse trabalho uma vez que a análise é realizada sobre a teoria utilizada nessas ferramentas e não em uma sua implementação específica. Cadeia de Markov é uma teoria probabilística onde o resultado de um experimento é influenciado pelo resultado dos de experimentos anteriores. A cadeia é composta por estados e transições e é utilizada para representar a dependência entre experimentos. Cada transição é rotulada com valores de probabilidade de tal forma que a soma dos valores de probabilidades das transições que partem de um mesmo estado é igual a 100%. A cadeia se inicia em um determinado estado e transita de um estado a outro de acordo com as probabilidades das transições. A transição de um estado para outro é chamada de passo. A cada passo a probabilidade da transição de alcançar o próximo estado é independente da probabilidade das transições anteriores (Grinstead and Snell, 2006).

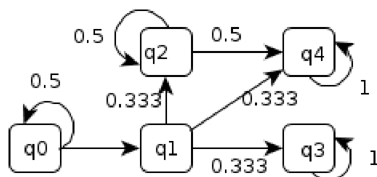


Figura 3.2: Exemplo de uma cadeia de Markov

Esses estados e transições podem também ser representados como um grafo direcionado cujas transições são rotuladas com probabilidades. Fig 3.2 apresenta um exemplo de de um grafo de uma cadeia de Markov. Os estados a partir dos quais não é possível sair são chamados de estados de estados absorventes. Na Fig 3.2, são apresentados dois estados absorventes: q4 e q3. Esses estados são considerados estados finais na cadeia e através deles é possível verificar na cadeia questões como:

- Qual a probabilidade de se alcançar q3 em algum momento? (tempo ilimitado)
- Qual a probabilidade de se alcançar q4 em dois passos? (tempo limitado)

Observe que as questões acima forma classificada em tempo ilimitado e limitado. Questões com tempo limitado são utilizada quando o número de passos feito deve ser limitado na cadeia de tal forma que apenas as transições que levam ao estado desejado dento do número limitado de passos seja consideradas. Por outro lado, questões com tempo ilimitado consideram todas as transições que de alguma forma levam ao estado desejado.

Cadeias de Markov podem ainda ser classificadas em tempo discreto (*Discrete Time Markov Chain*, DTMC) ou contínuo (*Continuous Time Markov Chain*, CTMC). CTMCs são modelos estocásticos onde as transições são feitas a determinada taxa ao invés de a uma determinada probabilidade (Hahn, 2008). Análise apresentada neste trabalho, assim como nos trabalhos relacionados, utilizada modelos DTMC (Ghezzi and Sharifloo, 2011b; Rodrigues et al., 2012).

A ferramenta PRISM especifica a linguagem PRISM: uma linguagem baseada em estados derivada do formalismo de Módulos Reativos e utilizada lógica temporal tal como

Lógica probabilística de computação ramificada (*Probabilistic Computational Tree Logic*, PCTL) para construir a cadeia de Markov e verificar propriedades no modelo (Baier and Katoen, 2008; Hansson and Jonsson, 1994; Bianco and Alfaro, 1995).

Com essa linguagem é possível modelar processos, que na linguagem PRISM são chamados módulos. Um modelo PRISM é composto por um ou mais módulos. Cada módulo possui um conjunto de variáveis com intervalo de valores finito que definem os possíveis estados desse módulo. O modelo final é a síntese de todos os módulos através de composição paralela. Cada módulo é comporto por um conjunto de comandos com guardas. Por exemplo, um comando DTMC em PRISM possui a seguinte forma:

Listagem 3.1: Comando PRISM

```
[ação] <guarda> -> <expressão> : <atualização de variável>;
```

Uma guarda é um predicada sobre todas as variáveis do modelo e uma vez satisfeito, o módulo fará a transição com uma certa probabilidade expressa por *expresso*, para atualizar o estado do modelo. um comando pode conter vários pares de <expressão> : <atualização de variável> representado as transições que deixam o estado atual, nesse caso cada par é separado por um símbolo '+'. Cada expressão pode envolver diversas constantes racionais e resultar em um número racional. A soma de todas as expressões em um único comando é um número racional p tal que $0 \leq p \leq 1$ que representa 0% e 100% de probabilidade respectivamente. A ação pode ser utilizada para rotular um comando que sincroniza com outro comando em um módulo diferente. Quanto não há rótulo de ação os comandos executarão assincronamente.

A ferramenta PRISM realiza *model checking* determinando o valor quantitativo de cada propriedade especificada e se o modelo as satisfaz. Nos exemplos apresentados nesse trabalho são utilizadas questões PCTL para verificara a probabilidade de se alcançar um estado final de sucesso de forma a estimar a confiabilidade do software que o modelo representa.

3.2.2 Model checking de LPS

Aplicar o mesmo processo em LPS não é viável uma vez que todos as etapas teriam de ser repetidos para cada diferente configuração.

Quando lidamos com LPS, é desejável construir um *único* modelo capaz de verificar a confiabilidade de todos os produtos. Entretanto, isso implica que a variabilidade deve ser tratada diretamente no modelo. Tais variabilidades podem ser tratadas no modelo reduzindo o esforço de se construir um modelo diferente para cada configuração. Entretanto,ainda será necessário realizar o *model checking* para cada configuração. Assim, um técnica que lide com esse problema trata variabilidade no modelo e permite verificar propriedades de diferentes configurações no mesmo modelo. Isso pode ser feito por meio de *model checking* paramétrico. Com parâmetros no modelo é possível mudar sua semântica (trocando valores de parâmetros) de tal forma que represente diferentes configurações. Fig 4.1 apresenta uma visão geral do processo para uma LPS. Observe que o processo é o mesmo, porém com alterações nas entradas e saídas. Em particular, destaca-se o *feature model* como entrada e a fórmula aritmética como saída final. Essas atividades são conduzidas por um engenheiro de domínio da LPS. A fórmula é composta por parâmetros definidos no modelo para representar a variabilidade e sua avaliação resulta no valor final

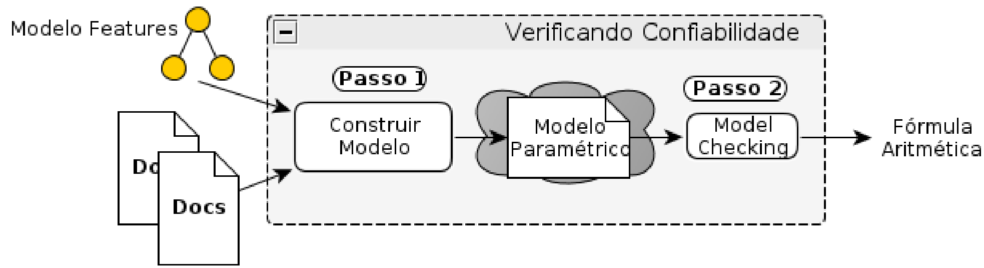


Figura 3.3: Processo de *model-checking* paramétrico

de confiabilidade para cada configuração da LPS. O engenheiro de aplicação da LPS, por sua vez, utiliza a fórmula para calcular a confiabilidade de uma configuração específica.

Variabilidade em modelos paramétricos pode ser tratada de diferentes maneiras. Modelos paramétricos são máquinas de estados; cada transição é rotulada com uma parâmetro de probabilidade ou um valor constante. É possível, por exemplo, tratar variabilidades rotulando transições com parâmetros cuja avaliação com diferentes valores altere a semântica do modelo. É possível também tratar variabilidade adicionando transições especiais rotuladas com parâmetros para saltar alguns estados de acordo com as diferentes avaliações. É possível ainda limitar a valoração dos parâmetros a um intervalo de valores válidos para um melhor controle do comportamento do modelo. Esses são apenas alguns exemplos do que pode ser feito em um modelo paramétrico para lidar com variabilidade dentro do modelo.

Qualquer que seja a escolha, a estratégia utilizada terá um impacto direto no tamanho da fórmula aritmética. Alguns trabalhos já destacaram esse aspecto. Alguns autores já alertaram que o uso excessivo de parâmetros no modelo pode fazer com que as ferramentas de fato não realizem o model checking e apenas apresentem como resultado uma fórmula que representa toda a computação da verificação (Hahn, 2008; Ghezzi and Sharifloo, 2011b). A análise realizada nesse trabalho, mostra que que escolhas erradas nas estratégias de modelagem podem levar a geração de fórmulas grandes.

Essas decisões são tomadas no Passo 1, apresentado na Fig 4.1 e esse passo pode ser manual, automático ou semi-automático, entretanto o passo 2 é praticamente apenas automático (apesar de poder ser realizado de maneira manual, isso não seria razoável). Esse trabalho detalha o processo de obtenção da fórmula aritmética a partir de um modelo paramétrico por meio da ferramenta PARAM (Passo 2) enfatizando as decisões de modelagem e relacionando-as com seu impacto no tamanho da fórmula de uma maneira quantitativa. Conhecer o impacto dessas decisões permite o desenvolvimento de técnicas mais abrangentes em relação aos tipos de variabilidade e que gerem fórmula com um tamanho esperado.

PARAM é uma ferramenta para model checking paramétrico probabilístico. De maneira similar à ferramenta PRISM, lida com modelos baseados em cadeias de Markov (CTMC, DTMC). Essa ferramenta utiliza uma variante da linguagem PRISM em que a principal diferença é a definição da palavra chave `param`. Essa palavra chave é utilizada para indicar que o valor de uma dada variável não é constante e não estará disponível durante o *parsing* do modelo.

É chamado de modelo PARAM os modelos que utilizam essa variante da linguagem PRISM. A ferramenta PARAM recebe como entrada um modelo PARAM e uma expressão

PCTL e produz como saída uma fórmula aritmética com os parâmetros definidos no modelo. Através da avaliação desses parâmetros é possível obter valores que respondam as consultas em um dado PCTL (Hahn et al., 2010).

No modelo param, a *expresso* em um comando (ver Listagem 3.1) pode conter parâmetros também. Essas expressões são *polinômios* cuja avaliação (através da avaliação dos parâmetros) é a probabilidade de transição p restrita ao mesmo intervalo de valores das transições de um modelo PRISM: $0 \leq p \leq 1$ (Greuel and Pfister, 2007). Essa característica é relevante e será revisitada posteriormente na análise.

A ferramenta PARAM sintetiza um autômato finito, extrai a expressão regular correspondente e, por fim, converte a expressão regular em uma fórmula aritmética.

3.3 Exemplo

Para melhor ilustrar os conceitos apresentados ao longo desse trabalho será introduzido um exemplo de um LPS e um possível modelo paramétrico que a representa.

Fig 3.4 apresenta um trecho de um modelo de *features* de um sistema de monitoramento de sinais vitais. Esse trecho é suficiente para ilustrar as ideias apresentadas nesse trabalho. Esse sistema consiste de um núcleo central e opcionalmente chama os componentes para monitoramento por meio do sensores EKG (eletrocardiógrafo) e/ou SPO2 (Saturação de oxigênio no sangue). Esses componentes são mapeados para as *features* `EKG` and `SPO2` respectivamente.

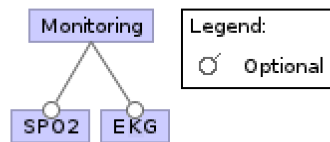


Figura 3.4: Modelo de *Features* do Sistema de Monitoramento de Sinais Vitais

Esse modelo de *features* possui quatro possíveis configurações, uma apenas com a *feature* EKG selecionada, outro apenas com a *feature* SPO2 selecionada, outro com ambas as *features* selecionadas e um apenas com a *feature* raiz selecionada. Este exemplo foi selecionada devido a sua expressividade. Observe que, o mesmo modelo de *features* da Fig 3.4 pode ser restrito por meio de *features* `OR` ou `Alternativas` , mas esses tipos de restrições levariam a um caso particular do exemplo com menos configurações.

Feature	**Artefatos**
MONITORING	Núcleo do sistema
EKG	Componente que trata os dados do sensor EKG
SPO2	Componente que trata os dados do sensor SPO2

Tabela 3.1: *Configuration Knowledge*

Fig 4.5(c)apresenta um diagrama de sequência que ilustra a configuração `{MONITORING,EKG, SPO2}`. Com a seleção das *features* de uma dada configuação e do CK (configuration knowledge, mapeamento entre artefatos e *features*) é possível construir o sistema

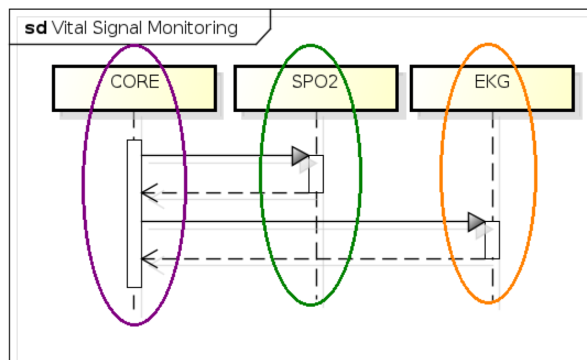


Figura 3.5: Configuração `{MONITORING,EKG,SPO2}`.

com três componentes: CORE, EKG and SPO2 (Czarnecki and Eisenecker, 2000). Note que a correspondência entre os componentes e as *features* é uma particularidade do exemplo. A Tabela 3.1 descreve o CK. Outras configurações tem um diagrama de sequência análogo diferindo apenas pela remoção de componentes.

3.4 Tratando expressividade

Esta seção apresenta uma abordagem para modelar variabilidade em um modelo PARAM (Passo 1 da Fig 4.1) e descreve o processo de obtenção da fórmula aritmética (Passo 2 da Fig 4.1).

Os principais passos da técnica serão descritos de maneria resumida de forma a permitir o entendimento da análise apresentada na Seção 3.5. O objetivo não é apresentar algoritmos, e sim o problema, através do qual é possível destacar características tais como a taxa de crescimento da fórmula. Fig 3.6 apresenta uma visão geral dos passos no processo de conversão. Primeiro, é feito o *parsing* do modelo paramétrico e construído seu correspondente autômato finito (Passo 2.1), então o autômato é reduzido de acordo com as restrições impostas pela expressão PCTL utilizada como entrada (Passo 2.2), a partir desse autômato é obtida sua correspondente expressão regular (Passo 2.3), que por fim é convertida em uma fórmula aritmética (Passo 2.4). Esse processo de conversão recebe duas entradas: um modelo PARAM e uma expressão PCTL.

Passos 2.1 and 2.2 are descritos in Seção 3.4.1, Passos 2.3 and 2.4 are descritos na Seção 3.4.2.

Para melhor explicar o processo, o exemplo da Seção 3.3 será expandido com seu correspondente modelo paramétrico. Note que, como mencionada na Seção 4.2, há diferentes formas de tratar variabilidade em um modelo PARAM.

Este trabalho apresenta uma abordagem de modelagem capaz de lidar comando *features* `Opcionais`. Os demais tipos de variabilidades (`OR, Alternativas, Obrigatórias`) podem ser transformadas em *features* `Opcionais` restritas por restrições *cross tree*, assim, os demais tipos são apenas restrições sobre a variabilidade `Opcional` (Gheyi et al., 2008). Dessa forma, esse trabalho propõe uma abordagem para modelagem de modelos paramétricos uma vez que as existentes não dão suporte à *features* `Opcionais` e poderiam limitar a análise apresentada (Ghezzi and Sharifloo, 2011b).

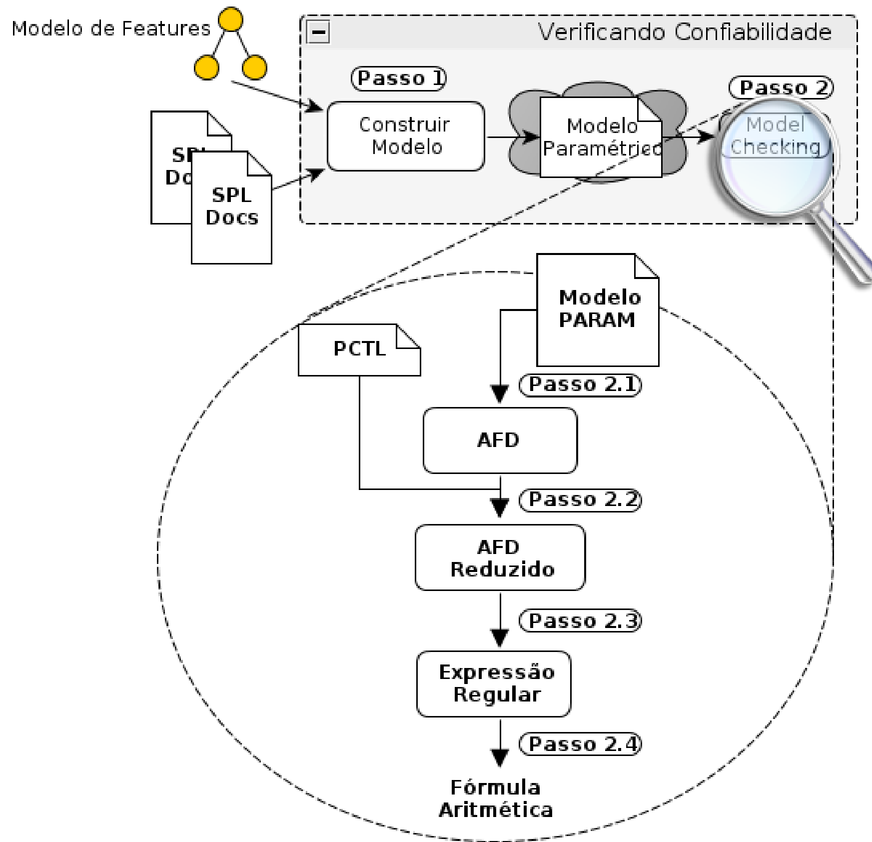


Figura 3.6: Visão Geral do Processo de Conversão

O processo de geração da fórmula é o mesmo qualquer que seja o modelo utilizado como entrada.

A estratégia de modelagem utilizada para modelar o exemplo da Seção 3.3 é guiado pelas seguintes regras:

1. Cada componente de software é mapeado para um módulo PRISM.
2. As transições do diagrama de sequência apontam para o componente que a executa.
3. Toda *feature* não obrigatória do modelo de *features* possui um parâmetro correspondente cujos valores válidos são 1 ou 0.
4. A variabilidade é tratada por meio de comando de desvio capaz de saltar os comandos relacionados a uma *feature* não selecionada utilizando seu parâmetro correspondente.
5. Cada passo tem uma chance de falha associada com o componente que o executa.

A Regra 1 é uma regra de conveniência uma vez que no exemplo, por simplificada, há uma correspondência entre *features* e componentes. A Regra 2 estabelece o relacionamento entre o diagrama de sequência e modelo PARAM. A Regra 3 assegura que apenas *features* que podem variar tem um parâmetro correspondente. A Regra 4 define como a variabilidade é tratada e é discutida posteriormente com mais detalhes. A Regra 5 define

a abordagem utilizada para calcular a confiabilidade da LPS uma vez que cada passo do software tem uma chance de falha associada.

Listagem 3.2: Modelo PARAM

```
dtmc
param int fSPO2;
param int fEKG;
const double rCORE = 0.999;
const double rSPO2 = 0.995;
const double rEKG = 0.997;

module Core
s0 : [0..8] init 0;
[] s0 = 0 -> (fSPO2*rCORE) : (s0'=1) +
                (1-fSPO2) : (s0'=3) +
                (fSPO2*(1 - rCORE)) : (s0'=7);
[SPO2] s0 = 1 -> (s0'=2);
[return_SPO2] s0 = 2 -> (s0'=3);
[fEKG_decision] s0 = 3 -> (fEKG*rCORE) : (s0'=4) +
                (1 - fEKG) : (s0'=6) +
                (fEKG*(1 - rCORE)) : (s0'=7);
[EKG] s0 = 4 -> (s0'=5);
[return_EKG] s0 = 5 -> (s0'=6);
[success] s0 = 6 -> (s0'=6); //END SUCCESS
[FAIL] s0 = 7 -> (s0'=7);  //END  FAIL
endmodule

module SPO2
s1 : [0..2] init 0;
[SPO2] s1 = 0 -> rSPO2 : (s1'=1) +
        (1 - rSPO2) : (s1'=2);
[return_SPO2] s1 = 1 -> (s1'=1);
[FAIL_SPO2] s1 = 2 -> (s1'=2);
endmodule

module EKG
s2 : [0..2] init 0;
[EKG] s2 = 0 -> rEKG : (s2'=1) +
        (1 - rEKG) : (s2'=2);
[return_EKG] s2 = 1 -> (s2'=1);
[FAIL_EKG] s2 = 2 -> (s2'=2);
endmodule
```

Constantes, declaradas com a palavra reservada const, prefixadas com a letra `r` representam a confiabilidade estimada de uma execução de um componente. Essas constantes são sufixadas com o nome de seu respectivo componente. O complemento desses valores, $(1 - rCORE)$ por exemplo, representa a chance de falha. Note que esses valores representam probabilidade, dessa forma o valor complementar é relacionado ao total de 100%.

As variáveis `s0`, `s1` e `s2` representam o estado do módulos que as contém no modelo PARAM. Mudanças nesses valores representam mudanças no estado do modelo de uma forma geral.

As Regras 3 e 4 são responsáveis por tratar a variabilidade. Isso é feito por meio da inserção de comandos de desvio antes de um comando que sincroniza sua execução com outro módulo que é mapeado para uma *feature*. O comando de desvio possui três transições: uma para a *feature* correspondente outro saltando para o primeiro comando após os comandos relacionados com a *feature* e outro representando a chance de falha. O modelo na Listagem 3.2) possui dois comandos de desvio: um para a *feature* SPO2 (linhas 10-12) e outro para *feature* EKg (linhas 15-17).Ambos possuem as três transições discutidas anteriormente. No comando de desvio para o SPO2 a avaliação do parâmetro

fSPO2 cujos valores válidos estão limitados a 0 ou 1 (Regra 3) seleciona entre a transição da linha 11 e o par de transições complementares na linha 10 e 12. Note que a transição da linha 11 é mutuamente exclusiva com o par de transições das linhas 10 e 12 uma vez que se a variável fSPO2 é valorada com 1 ela desabilita a transição da linha 11 associando 0% de probabilidade a ela e se essa variável for valorada com 0 as transições das linhas 10 e 12 serão simultaneamente desabilitadas com 0% de probabilidade para a ocorrência dessas transições e habilitará a transição da linha associando 100% de probabilidade de ocorrência da transição. O par de transições das linhas 10 e 12 representam as transições comuns no modelo enquanto a transição da linha 11 é utilizada para saltar os comandos relacionados à *feature* SPO2. O funcionamento é análogo para a *feature* EKG cujo comando de desvio está na linha 15. Na Seção 3.5 serão apresentadas as características inerentes ao *model checking* de modelo probabilístico paramétrico e à ferramenta PARAM.

3.4.1 Do modelo Paramétrico para o AFD

A linguagem PARAM é baseada em um formalismo de componentes concorrentes que permite a representação de componentes síncronos e assíncronos de forma modular. Essa linguagem provê abstrações sobre uma máquina de estados que permite o uso de conceitos de alto nível como módulos e variáveis (Alur and Henzinger, 1996). As transições nessas máquinas de estado podem ser rotuladas com probabilidades. Essas máquina de estados rotuladas com probabilidades são modeladas como cadeias de Markov.

Inicialmente, a ferramenta PARAM faz o *parsing* do modelo e constrói a cadeia de Markov correspondente (Passo 2.1 na Fig 3.6). A cadeia de Markov gerada segue a definição de um autômato finito determinístico AFD (Hopcroft et al., 2006):

$A = (Q, \sum, \delta, q_0, F)$

- Q é o conjunto de estados.
- $\sum$ é o conjunto de símbolos de entrada, ou alfabeto.
- δ é a função de transição ($\delta : Q \times \sum \rightarrow Q$)
- q_0 é o estado inicial.
- F é o conjunto de estados finais.

Onde Q é um conjunto de estados de uma máquina de estados, $\sum$ é o conjunto composto por todas as expressões que são rotuladas por transições do modelo PARAM, , δ define as transições entre os estados, q_0 é o estado inicial do modelo e F é o conjunto de todos os estados absorventes do modelo PARAM.

O processo de conversão é principalmente baseado nas seguintes regras (Alur and Henzinger, 1996):

- Um estado é uma valoração de todas as variáveis no modelo. Cada valoração diferente representa um estado diferente.
- Um guarda de sincronização implica que uma ou mais variáveis trocam de valores simultaneamente. Assim, em uma única troca de estados uma ou mais variáveis trocam de valores.

Note que cada rótulo de transição é apenas um *token*, um símbolo no alfabeto $\sum$, mesmo as expressões complexas envolvendo expressões, constantes e parâmetros. Essa expressão não pode ser operada com outras expressões enquanto for considerada um *token* do AFD. Para tornar a apresentação mais clara, utilizou-se a substituição de variáveis apresentada na Tabela 3.2. Essa tabela apresenta os *tokens* utilizados em substituição às expressões que ocorrem no modelo e as linhas da Listagem 3.2 em que essas expressões ocorrem.

Token	**Expressão**	**Linhas**
a	(fSPO2*0.999)	10
b	(1-fSPO2)	11
c	(fSPO2*0.001)	12
d	(fEKG*0.999)	15
e	(1 - fEKG)	16
f	(fEKG*0.001)	17
g	0.995	26
h	0.005	27
i	0.997	34
j	0.003	35

Tabela 3.2: *Substituição de Variáveis*

Essa substituição de variáveis será revisitada em seções posteriores. Fig 3.7 ilustra o AFD obtido a partir do modelo da Listagem 3.2. Cada estado é rotulado por uma tupla (s0,s1,s2)que representa a valoração das variáveis s0, s1 and s2 do modelo.

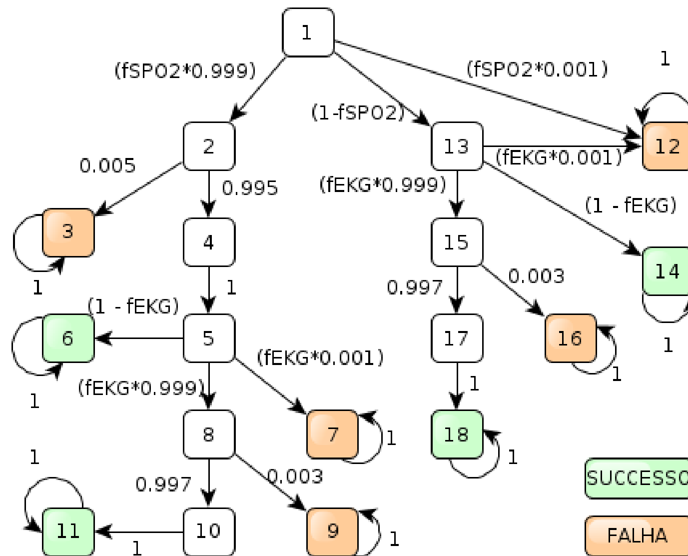


Figura 3.7: AFD da Listagem 3.2

O passo 2.2 na Fig 3.6 consiste em fazer o *parsing* da expressão PCTL e eliminar do AFD os estados que nunca serão visitados em qualquer caminho a partir do estado inicial até algum estado final definido pela expressão PCTL.

O objetivo é calcular a confiabilidade de todos os produtos da LPS. Assim, o objetivo é obter a probabilidade de diferentes configurações alcançarem o estado final de sucesso. Como apresentado na Listagem 3.2, a linha 20 representa o estado final de sucesso para todas as configurações. Logo, deseja-se calcular a probabilidade de se alcançar algum estado onde variável $s0$ do modelo esteja valorada com 6. A seguinte expressão PCTL representa essa questão:

$$P \;=?\; [\; true \; U \; s0 = 6 \;] \tag{3.1}$$

Esse PCTL é utilizado para determinar os estados finais de sucesso e os caminhos que levam a esses estados. Com esses caminhos é possível reduzir o AFD removendo os estados que nunca alcançarão nenhum estado final. Neste trabalho um caminho é definido como em teoria dos grafos (Bondy and Murty, 2008).

No exemplo da Fig 3.7 o estados finais são:

$$F \;= (6,0,0),(6,0,1),(6,1,0),(6,1,1) \tag{3.2}$$

Qualquer estado que não pode alcançar algum desses estados pode ser removido do AFD. O estados abaixo podem ser removidos do AFD apresentado na Fig 3.7:

$$(2,2,0),(5,0,2),(5,1,2),(7,0,0),(7,1,0) \tag{3.3}$$

Essa redução conclui o Passo 2.2 na Fig 3.6. Para computar o valor descrito pelo PCTL é preciso identificar cada caminho a partir do estado inicial para algum estado final do AFD reduzido. Cada caminho é composto por uma sequência de transições e seus rótulos correspondentes. Os valores desses rótulos são multiplicados para se obter a probabilidade de se alcançar o estado final a partir do estado inicial do caminho. Somando-se as probabilidades de todos caminhos identificados pelo PCTL obtemos a probabilidade final da consulta expressa pelo PCTL (Kwiatkowska et al., 2007). A Seção 3.4.2 detalha o processo de obtenção desse valor a partir da expressão regular correspondente do AFD reduzido.

3.4.2 Do AFD para a Fórmula

O Passo 2.3 na Fig 3.6 consiste em obter a expressão regular correspondente a partir do AFD reduzido. Essa expressão regular é utilizada para computar a fórmula final aritmética como proposto por (Daws, 2005). Expressões regulares definem a mesma classe de linguagens que um AFD. Assim, um AFD possui uma expressão regular correspondente e vice versa (Hopcroft et al., 2006).

O algoritmo de eliminação de estados pode ser utilizado para converter um AFD para uma expressão regular. Neste trabalho foi utilizada a ferramenta JFLAP para modelar o AFD e computar sua expressão regular (Rodger, 2012). A expressão regular obtida a partir do DFA reduzido é:

$$be1 * +ag1e1 * +ag1di11 * +bdi11* \tag{3.4}$$

Onde '$*$' é o fecho de Kleene e '$+$' é o operador de união e a concatenação é o operador (implícito) definido por dois *tokens* consecutivos.

Essa expressão é regular é convertida em uma fórmula aritmética utilizando a seguinte definição recursiva (Hahn, 2008):

1) $val\binom{p}{q} = \frac{p}{q}$	4) $val(rs) = val(r).val(s)$
2) $val(x) = x, x \in \sum$	5) $val(r^*) = \frac{1}{1-val(r)}$
3) $var(r+s) = val(r) + val(s)$	

Onde p e q são números racionais, r, s são *tokens* e x são variáveis.

Note que a regra 5 é definida apenas quando $0 \leq r < 1$, se $r = 1$ então $val(r^*) = 1$. Essas regras devem ser aplicadas na ordem de precedência da definição acima. Ao aplicar essas regras obtém-se a seguinte fórmula para o exemplo apresentado:

$$b * e + a * g * e + a * g * d * i + b * d * i \tag{3.5}$$

Ao desfazer a substituição de variáveis definidas na Tabela 3.2 obtém-se:

$$\begin{aligned}(1 - fSPO2) * (1 - fEKG) + (fSPO2 * 0.999)\\ *0.995 * (1 - fEKG) + (fSPO2 * 0.999) * 0.995\\ *(fEKG * 0.999) * 0.997 + (1 - fSPO2)*\\ (fEKG * 0.999) * 0.997\end{aligned} \tag{3.6}$$

A equação 3.6 é a mesma fórmula obtida pelo PARAM para o modelo da Listagem 3.2 diferindo apenas de simplificações. A fórmula final gerada pela ferramenta PARAM é a seguinte:

$$\begin{aligned}(4792403 * fSPO2 * fEKG - 1199000000 * fSPO2\\ -799400000 * fEKG + 200000000000)/(200000000000)\end{aligned} \tag{3.7}$$

Com isso, conclui-se o Passo 2.4 da Fig 3.6 e completa o processo de conversão de um modelo paramétrico representando configuração de uma LPS em uma fórmula aritmética.

De acordo com a estratégia utilizada para modelar a LPS, a fórmula possui dois parâmetros diferentes: $fSPO2$ and $fEKG$. Esses parâmetros podem ser valorados com 0 ou 1 representando a não seleção e seleção de *features* respectivamente. O parâmetro $fSPO2$ é usado para selecionar a *feature* $SPO2$ e o parâmetro $fEKG$ is usado para selecionar a *feature* EKG. A Seção 3.5 discute aspectos desse processo e destaca os principais aspectos relacionados com o tamanho da fórmula.

3.5 Análise de Escalabilidade

O tamanho da fórmula é fortemente relacionado ao AFD. Este trabalho faz uma avaliação analítica dos rótulos do DFA e de como estes impactam o tamanho da fórmula e do tamanho da expressão regular e como isso impacta o tamanho da fórmula e relaciona esses aspectos com as estratégias de modelagem.

Este trabalho apresenta resultados obtidos a partir de uma simulação de exemplo apresentado expandido para conter mais *features*. Essa simulação provê uma motivação de o quão rápido o tamanho da fórmula pode crescer com o número de *features*. Além disso, serão discutidas algumas implicações práticas relacionadas ao tamanho da fórmula enfrentadas no contexto de um projeto de pesquisa.

Para analisar o tamanho da fórmula é preciso definir como o tamanho da fórmula será medido. Neste trabalho, o tamanho da fórmula é medido de acordo com a quantidade de operandos. Apesar dessa medida não considerar os custos de avaliação de diferentes

operações, ela provê uma dimensão no custo de avaliação que pode ser majorada (caso hjam apenas multiplicação , divisão e potenciação) ou minorada (caso hajam apenas adições e subtrações), e provê uma noção do custo de avaliação da fórmula uma vez que fórmulas grandes possuem um grande número de caracteres e bytes. Utilizando essa métrica, as fórmulas na Seção 3.4.2 tem os seguintes valores: Fórmula 3.5 possui tamanho 12, Fórmula 3.6 possui tamanho 20 e Fórmula 3.7 possui tamanho 9. Todas as fórmulas referencias nessa seção se referem a fórmula apresentadas na Seção 3.4.2.

3.5.1 Avaliação Analítica

Esta seção analisa aspectos que impactam diretamente o tamanho da fórmula. Inicialmente será discutido sobre os rótulos em comandos de um modelo PARAM e como estes podem fazer a fórmula crescer, em seguida será apresentada a correspondência entre a expressão regular e as estratégias de modelagem.

No exemplo apresentado, a fórmula obtida por meio da expressão regular possui doze operandos e a fórmula final obtida pela ferramenta PARAM possui apenas nove. Isso corre por que alguns *tokens* foram substituídos por expressões com um ou mais operandos, alguns deles constantes que foram somadas em uma única constante. O pior cenário ocorre quando os *tokens* são expressões com adições e subtrações de parâmetros, uma vez que os parâmetros não podem ser operados. Por exemplo, suponha que o *token* 'b' seja substituído por '$x + y$' e 'e' por '$z - w$' na Fórmula 3.5 e que a propriedade distributiva seja aplicada, assim teremos:

$$\begin{aligned} x*z - x*w + y*z - y*w + a*g*z - a*g*w \\ +a*g*d*i + x*d*i + y*d*i \end{aligned} \tag{3.8}$$

Com essa substituição o tamanho da fórmula sobre para 24.

Considere a Fórmula 3.5. Esta possui seis *tokens* diferentes (a,b,d,e,g,i), que vem da substituição de variáveis. Vale ressaltar, que a estratégia de substituição de variáveis foi adotada para facilitar o entendimento do conceito de *token* durante o processo. Suponha que cada *token* é substituído por um parâmetro diferente. Nessa caso, quaisquer que sejam os operados na fórmula, esta não teria menos do que seis operandos, uma vez que os parâmetros não podem ser operados.

O número de parâmetros afeta diretamente o tamanho da fórmula dado que, se não houvesse nenhum parâmetro fórmula poderia sempre ser simplificada para um único valor numérico e por outro lado, se a fórmula fosse composta apenas por fórmulas esta não poderia ser simplificada facilmente.

Cada *token* na expressão regular é uma substituição para uma expressão na Tabela 3.2. Essas expressões são provenientes de transições do modelo PARAM. Note que, cada comando em um modelo PARAM é composto por uma ou mais transições e cada transição é rotulada por um *polinômio* que representa sua probabilidade. Polinômios são compostos por *termos* e termos são definidos como multiplicação de constantes e de zero ou mais variáveis (Greuel and Pfister, 2007). Neste trabalho, essas variáveis são parâmetros do modelo PARAM. Um polinômio pode ser expressado em um produto de somas ou em uma soma de produtos.

No exemplo desse trabalho, algumas transições são rotuladas com soma de termos, por exemplo `(1 - rCORE)`. Transições sequênciais geram multiplicações, esse resultado vem

do item 4 da definição recursiva apresentada na Seção 3.4.2. Assim, transições sequênciais rotuladas por somas de termos geram produtos de somas. A ferramenta PARAM aplica a propriedade distributiva numa tentativa de simplificar a fórmula final. De forma geral, se houver mais contantes do que parâmetros esta heurística resulta em um menor número de operandos uma vez que as contantes racionais podem ser somadas. Como mostrado anteriormente esse produto pode aumentar o número de termos, e consequentemente o número de operandos, a medida em que se aumenta o número de diferentes parâmetros.

A estrutura da fórmula final é obtida a partir da expressão regular correspondente do AFD. A expressão regular obtida por meio da conversão do AFD através do algoritmo de eliminação de estados pode diferir em tamanho e formato dependendo da ordem em que os estados são eliminados (Ahn and Han, 2009). Essa expressão regular não pode ser minimizada eficientemente (Gramlich and Schnitger, 2007). Isso dificulta a estimativa do tamanho da expressão regular.

Entretanto é possível utilizar um limite superior do tamanho da expressão regular correspondente. Apesar do limite superior representar o cenário de pior caso, ele provê uma relação entre atributos do AFD original e o tamanho de sua expressão regular correspondente.

O tamanho da expressão regular pode ser definido de diferentes maneiras, mas o número de símbolos alfabéticos é considerada a métrica mais útil (Ellul et al., 2004). Essa métrica é equivalente à métrica definida para o tamanho da fórmula. Símbolo alfabético é qualquer símbolo que pertence à $\sum$ (Seção 3.4.1). Com essa medida de tamanho da expressão regular temos o seguinte limite superior:

$$|Q| * |\sum| * 4^{|Q|} \tag{3.9}$$

Onde $|Q|$ é o número de estados e $|\sum|$ é o tamanho do alfabeto. Esse limite superior indica que o número de estados impacta exponencialmente no pior caso e que o tamanho do alfabeto afeta linearmente o tamanho da expressão regular.

Observe que o alfabeto é composto por *tokens*, ou seja, cada expressão na Tabela 3.2 é um elemento diferente do alfabeto. O número de parâmetros afeta diretamente o tamanho da fórmula uma vez que este aumenta o tamanho do alfabeto. Os parâmetros podem aumentar ainda mais o tamanho do alfabeto uma vez que estes podem ser utilizados em expressões que são consideradas *tokens* distintos.

Para mitigar o tamanho da fórmula final, é preciso observar os pontos discutidos anteriormente. A enumeração abaixo os resume e os relaciona com a modelagem:

1. O número de estados do modelo deve ser reduzido.

2. O número de parâmetros em expressões que rotulam transições deve ser reduzido.

3. O número de parâmetros deve ser reduzido.

Observe que o Item 1 é a variável mais crítica, pois afeta diretamente e exponencialmente o tamanho da fórmula. O Item 2 pode ser crítico a medida em que que o número de transições sequênciais rotuladas por polinômios aumenta. O Item 3 é a variável menos crítica com impacto linear, entretanto forma como é utilizado no modelo pode ser crítica, por exemplo, utilizando expressões com parâmetros.

As regras utilizadas para construir o modelo PARAM do exemplo utiliza comandos de desvio para pular transições. Essa estratégia é conflitante com o Item 1 uma vez que a adição de comandos adicionais para tratar variabilidades insere estados desnecessários do ponto de vista da documentação do sistema para cada parte do sistema que interage com componentes relacionados com *features*. Essa estratégia também é conflitante com o Item 2 uma vez que em cada um desses pontos é incluída uma transição rotulada com expressões envolvendo parâmetros (`1-fSPO2`, por exemplo). Essas soluções devem ser evitadas.

Ghezzi et al. trata variabilidade utilizando o mesmo parâmetro para selecionar a *feature* e representar sua confiabilidade, e o mesmo parâmetro pode ser utilizado para representar uma ou mais *features*. Um único parâmetro pode ser utilizado para selecionar *features* pertencentes a um mesmo ponto de variação, assim, apenas um parâmetro por ponto de variação é necessário.

Mas esse abordagem é restrita a *features* `Alternativas`. Pontos de variação apenas com *features* `Alternativas` possuem uma característica particular: esse tipo de ponto de variação é sempre preenchido com uma e apenas uma *feature* em uma dada configuração (Czarnecki and Eisenecker, 2000). Utilizando essa característica o modelo é construído de tal forma que o componente pode representar diversas *features* `Alternativas` (há uma premissa que cada *feature* é mapeada para apenas um componente de software). A seleção de *features* é definida pela valoração de parâmetros de pontos de variação com o valor de confiabilidade correspondente à *feature* que o componente representa. Essa estratégia de modelagem é eficiente considerando os três itens que impactam no tamanho da fórmula apresentados nessa seção.

Essa estratégia de modelagem pode ser estendida para cobrir pontos de variação com *features* `Or` e `Opcionais`, mas primeiramente é preciso introduzir alguns conceitos relacionados a pontos de variação (Czarnecki and Eisenecker, 2000):

- Os pontos de variação podem ser classificados em *singular* caso permitam que mais de uma *feature* relacionada ao ponto de variação ocorre em uma configuração da LPS
- Pontos de variação também podem ser classificados como *não singular* caso não permitam que mais de uma *feature* relacionada ao ponto de variação seja incluída em uma configuração.

Os pontos de variação VP são também classificados com relação ao seu tamanho:

- $size(VP)$: O número máximo de *features* que podem ser selecionadas para resolver o ponto de variação de um modelo de *features*

Todos os pontos de variação singulares podem ser tratados de maneira similar aos pontos de variação alternativos. Se o ponto de variação pode ser vazio em uma configuração, ou seja, se não é necessário selecionar nenhuma *feature* para preenchê-lo, o valor de confiabilidade 1 é utilizado. Apesar de o modelo PARAM ter um módulo para uma *feature* que não foi selecionada, sua avaliação com 1 o tornará neutro para a fórmula. Assim, um ponto de variação singular pode ser tratado com apenas uma variável.

Pontos de variação não singulares podem ser tratados de maneira simples. É preciso um número de variáveis e módulos no modelo PARAM igual a seu tamanho. Assim, é possível lidar com configurações que selecionem todas as *features* de um ponto de variação e

qualquer combinação por meio da substituição de valores de confiabilidade de *features* não selecionadas na configuração por 1 de maneira similar aos pontos de variação singulares.

Apesar dessa abordagem superar as limitações relacionadas a variabilidades do modelo de *feature* esta mantém restrições como por exemplo a correspondência entre *features* e componentes. Além disso, essa abordagem não considera o comportamento interno de um componente que é mapeado para uma *feature*.

Como trabalho futuro, será proposta uma abordagem de modelagem objetivando superar as premissas relacionadas à arquitetura, ao CK e à restrições de variabilidades. Essas abordagens compreendem o Passo 1 da Fig 4.1.

Essa análise demonstrou que estratégias envolvendo comandos de desvio e/ou expressões levam ao crescimento do tamanho da fórmula e devem ser evitadas. Além disso foram propostas diretrizes que podem ser utilizadas para estender as abordagens existentes.

3.5.2 Avaliação Prática

Esta seção discute, a partir de um ponde de vista prático, como a fórmula pode crescer com o número de *features* no exemplo da Seção 3.4 e exemplifica obstáculos enfrentados ao se tentar avaliar fórmulas grandes em tempo de execução.

Essa simulação mostra a taxa de crescimento da fórmula em número de operandos. Como foi discutido anteriormente, a estratégia de modelagem do exemplo apresentado pode levar a fórmulas grandes.

Essa simulação consiste em aumentar a variabilidade do modelo de *features* da of Fig 3.4 gradualmente por meio da adição de novas *features* . A Fig 3.8 ilustra o crescimento do modelo de *features*. Nessa figura foi adicionada uma a nova *feature* `ACELEROMETER`. A *feature* `SENSOR_N` representa a n-ésima adição de *feature*. As novas *features* adicionadas são também `Opcionais` de forma a reter a expressividade como discutido na Seção 3.3.

Novas *features* representando novos sensores foram inseridas gradualmente no exemplo de sistema de monitoramento de sinais vitais.

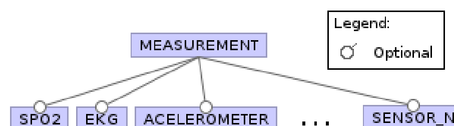


Figura 3.8: Extensão do modelo de *features*

A simulação é composta de várias iterações. Em cada iteração uma nova *feature* é adicionada ao modelo de *features*, um novo componente correspondente é adicionado ao diagrama de sequência e um novo módulo é acrescentado ao modelo PARAM. A Fig 3.9 apresenta o diagrama de sequência onde o componente `SENSOR_N` representa o n-ésimo componente correspondente à *feature* adicionada.

A Fig 3.10 apresenta um curva definida pelo número de *features* e o tamanho final da fórmula aritmética em número de operandos para cada iteração. Esse gráfico ilustra o quão rápido o tamanho da fórmula pode crescer com o número de *features*. As regras de modelagem utilizadas são fortemente relacionadas com essa taxa de crescimento. Essa taxa de crescimento pode ser ainda mais severa para modelos complexos. Aplicando as

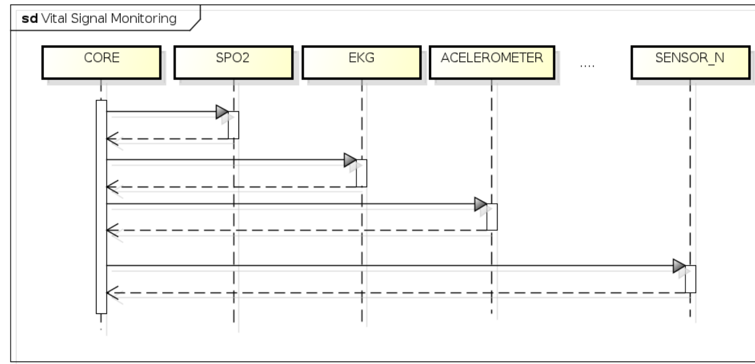


Figura 3.9: Extensão da documentação da LPS

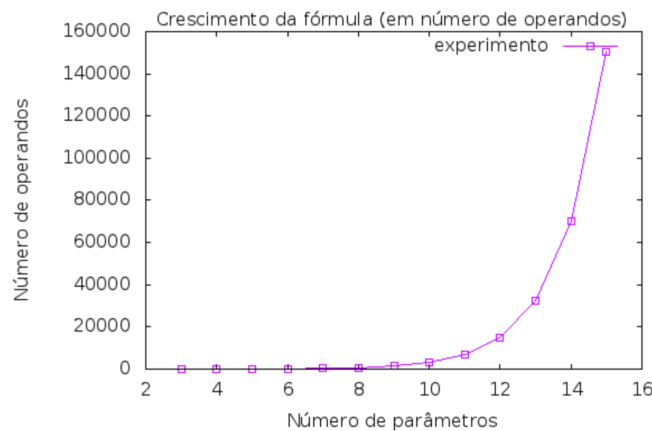


Figura 3.10: Crescimento da Fórmula com o Aumento da Variabilidade

mesmas regras para um exemplo complexo foram obtidos, para um modelo com pouco mais de setenta estados, uma fórmula com dezenas de milhões de operandos.

Fórmulas grandes tem implicações críticas relacionadas a sua avaliação. Um fórmula precisa ser escrita diretamente no código ou informada como entrada para um programa de forma a ser avaliada. Apesar de os computadores atuais lidarem com grande quantidades de operações por segunda, a fórmula precisa ser carregada e avaliada o que aumenta o uso de operações de I/O e o uso de memória.

Nosso projeto de pesquisa [2] consiste em um sistema de monitoramento de sinais vitais utilizando LPS. Esse sistema executa na plataforma Android e é desenvolvido em Java para Android. Seu modelo de *features* é composto por 12 *features*: 6 `OR`, 2 `Alternativas` e 4 `Obrigatórias`.

Utilizando o método apresentado na Seção 3.4, foi obtida uma fórmula com 259.470 operadores. Devido a uma limitação da máquina virtual Java em relação à quantidade de código por método (65536 bytes), foi preciso dividir a fórmula em vários métodos e chamá-los sequêncialmente para calcular o valor de confiabilidade (Oracle, 2011). Também foram feitos testes com bibliotecas existentes para avaliações de fórmulas [3]. Entretanto, avaliar fórmulas tão grandes por meio dessas bibliotecas é um processo ainda mais custoso.

[2]Ambient Assisted Living Product Lines - Projeto do CNPq

[3]Bibliotecas tais como: http://code.google.com/p/arity/ e http://projects.congrace.de/exp4j/

3.6 Trabalho Relacionado

Algumas contribuições lidam diretamente com o problema de *model checking* de LPS, porém todos eles possuem algum tipo restrição sobre a variabilidade e/ou arquitetura do software.

Ghezzi et al. propõe o uso de *model checking* paramétrico para verificar propriedades de uma LPS. Sua abordagem possui limitações sobre a arquitetura da LPS tais como cada *features* deve ser restrita a um módulo e, provavelmente a restrição mais crítica, o modelo de *features* deve conter apenas *features* alternativas (Ghezzi and Sharifloo, 2011b,a). Por outro lado, essa estratégia de modelagem é eficiente de acordo com a análise apresentada neste trabalho uma vez que esta não cria nenhum estado adicional nem utiliza-se de expressões envolvendo parâmetros.

Classen et al. propõem uma modelo de máquina de estados onde o comportamento do sistema (transições) é anotado com a *feature* relacionada (Classen et al., 2011, 2010). Esse trabalho apresenta um modelo especificamente projetado para LPS e trata o conceito de *features* nativamente. Essa abordagem estabelece um forte acoplamento entre o modelo de *features* e a arquitetura do sistema no modelo, uma vez que o fluxo do sistema deve ser relacionado com *features* isso dificulta a representação de entrelaçamento e espalhamento. Além disso, não permite a um comportamento (transição) no modelo estar relacionado com uma ou mais *feature*. Ademais, o *model checking* desses modelos não provê a flexibilidade de uma fórmula aritmética da abordagem de model checking paramétrico.

3.7 Conclusão

Estimar confiabilidade de software é uma tarefa importante, especialmente, para sistemas críticos. Essa tarefa se torna ainda mais difícil quando se trata de LPS. Apesar de alguns trabalhos tratarem diretamente esse problema, estes apresentam limitações como as discutidas na Seção 3.6.

Esse trabalho trata o problema de realizar *model checking* paramétrico sobre uma LPS. Foi apresenta o processo de *model checking* de uma LPS e apresentada uma visão detalhada do *model checking* paramétrico da ferramenta PARAM desde a documentação da LPS até a fórmula aritmética. Foi apresenta uma estratégia para lidar com as limitações de expressividade destacando suas implicações em termos de escalabilidade. Este trabalho focou em visão teórica do problema e portanto não apresenta algoritmos e otimizações implementadas pela ferramenta PARAM.

A grande lacuna desse processo é o Passo 1 na Fig 4.1. Abordagens atuais possuem limitações sobre a arquitetura do software e/ou variabilidade. O tamanho da fórmula pode crescer rapidamente com o número de *features* se a abordagem de modelagem não considerar os aspectos que impactam em seu tamanho. Por meio da explanação de conceitos e *trade-offs* envolvidos em um *model checking* paramétrico, é possível desenvolver novas técnicas para verificar propriedades, tais como confiabilidade, em LPS. Utilizando os resultados desse análise essas abordagens podem fazer menos restrições sobre a variabilidade por meio da administração dos *trade-offs* entre expressividade e o tamanho da fórmula cientes de quais decisões são mais críticas para a técnica de *model checking* paramétrico.

Capítulo 4

Uso de *Features* Opcionais em *Model Checking* paramétrico de LPS para análise de confiabilidade

A construção de software dependável é um tema relevante e desafiador, principalmente quando os softwares tratam de domínios críticos. Lidar com esse tipo de problema em Linhas de Produtos de Software (SPL) é ainda mais desafiador dada a característica de que o número de configurações possíveis cresce exponencialmente com o número de *features* presentes na LPS.

Recentemente, novas abordagens têm sido propostas para tratar desse problema. Em particular, abordagens que utilizam *model checking* paramétrico surgiram como soluções promissoras para avaliar confiabilidade. Entretanto, as estratégias atuais não tratam de todos os tipos de variabilidade possíveis dentro da LPS. Para preencher essa lacuna, este trabalho propõem uma abordagem válida (*sound*) para analisar propriedades da dependabilidade em LPS utilizando *features* opcionais. Como segunda contribuição, é feita uma avaliação sobre os conceitos básicos que auxiliam o desenvolvimento desta proposta.

Utilizando a esta proposta, é possível realizar análise de confiabilidade na LPS desde dos estágios de concepção de uma forma abrangente e válida.

4.1 Introdução

Garantir a dependabilidade de software, ou seja, garantir que o software possui níveis adequados de disponibilidade, confiabilidade, seguranÃğa (*security*), integridade e manutenibilidade é importante quando tratamos de domínios críticos, desde que falhas nesses sistemas podem impactar e desencadear consequências críticas (Avizienis et al., 2004).

Em particular, confiabilidade em termos do atributo de dependabilidade que expressa continuidade da operação correta do software, é uma propriedade fundamental neste contexto. Quantificar essas propriedades nas fases iniciais no ciclo de desenvolvimento de software evitar custos desnecessários com retrabalho, manutenção e evolução (Hoffman, 2008).

Model checking é uma técnica para estimar propriedades não-funcionais do sistema através da documentação do software, particularmente aquela que descreve o compor-

tamento de software (Rodrigues et al., 2012; Uchitel et al., 2004; Ghezzi and Sharifloo, 2011a).

Entretanto, o problema de verificação de propriedades no contexto de LPS é mais difícil do que nas técnicas tradicionais da engenharia de software. LPS é um paradigma da engenharia de software que visa o desenvolvimento de software via o reÃžso sistemático. Uma família de produtos é construída por um conjunto de artefatos com comunalidades e variabilidades expressos como *feature*, que é considerada uma unidade de diferenÃğa entre os produtos (Jilles Van et al., 2001).

Dado que uma LPS é um conjunto de famílias de produtos, verificar um modelo para cada produto da LPS é inviável, dado o crescimento exponencial no nÃžmero de configurações com o aumento da variabilidade e o reÃžso potencial de partes do modelo.

Por outro lado, o *model checking* paramétrico é uma abordagem promissora para quantificar propriedades de dependabilidade na LPS, desde que permite a resolução tardia da variabilidade do software. Ou seja, é possível avaliar o valor de dependabilidade somente quando os valores numéricos de cada configuração é provido (Hahn, 2008).

Ghezzi et al. propõem o uso do *model checking* paramétrico para verificar propriedades do software da LPS. Porém, essa proposta trata apenas de *features* alternativas (Ghezzi and Sharifloo, 2011b). Classen et al. propõem um modelo de máquina de estados onde o comportamento do sistema (transições) são anotadas e relacionadas às *features* (Classen et al., 2011, 2010). Este trabalho apresenta um modelo adaptado para LPS que trata os conceitos de *feature* nativamente, mas cria um acoplamento entre o modelo de *features* e a arquitetura do sistema no modelo, desde que o fluxo do sistema deve ser relacionado às *features*.

Além disso, o *model checking* deste modelo não provê a flexibilidade de uma fórmula aritmética da abordagem que utiliza *model checking* paramétrico. Por outro lado, o *model checking* paramétrico não permite a mudanÃğa da propriedade verificada no modelo uma vez que a fórmula é calculada.

Um modelo abrangente para tratar essa questão ainda é um problema em aberto, dado que não existe proposta conhecida capaz de quantificar propriedades da dependabilidade de um LPS utilizando qualquer tipo de variabilidade.

Este trabalho trata essa limitação através de uma metodologia que é capaz de representar qualquer tipo de variabilidade no *model checking* paramétrico através do uso de *features* opcionais (Seção 4.3).

Este capítulo possui as seguintes contribuições: 1) Demonstração de que utilizando *features* opcionais é possível representar outros tipos de variabilidade no modelo, 2) consistentemente realizar uma análise quantitativa de uma configuração de qualquer LPS. 3) Prova sistemática da validade da abordagem proposta para modelos gerados para LPS utilizando o *model checking* paramétrico (Seção. 4.4).

4.2 *Model Checking* paramétrico

Esta seção brevemente introduz conceitos basicos necessários para o entendimento do capítulo. Em especial, descreve conceitos relacionados à abordagem de *model checking*.

Este trabalho apresenta uma abordagem para modelagem baseada nas ferramentas probabilísticas para *model checking* tais como PRISM e PARAM. PRIS é uma ferramenta

probabilística para *model checking* baseada em cadeias de Markov. (Kwiatkowska et al., 2011).

Cadeias de Markov são sistemas de transições onde cada estado representa um evento probabilístico independente do evento anterior. Cada estado tem uma transição de chegada e n transições de saída que são rotuladas com a probabilidade de ocorrer. A principal propriedade da cadeia de Markov é a de *reachability* que representa a probabilidade de alcançar um estado a partir de outro em um determinado tempo de execução.

O tempo na cadeia de Markov pode ser representado de forma contínua ou discreta. No modelo de tempo discreto (Cadeia de Markov com Tempo Discreto ou ***D**iscrete **T**ime **M**arkov **C**hains* - DTMC), cada transição entre os estados representa que uma unidade de tempo aconteceu . Assim como outros trabalhos, este trabalho foca em DTMC (Ghezzi and Shariﬂoo, 2011b; Bianco and Alfaro, 1995).

Uma expressão PCTL (Lógica probabilística de computação ramificada ou ***P**robabilistic **C**omputational **T**ree **L**ogic* - PCTL) pode ser usada para consultar a probabilidade de alcançar um estado em um tempo determinado. Com expressões PCTL, é possível verificar o modelo e responder questões tais: *Qual é a probabilidade de eventualmente alcançar um estado de sucesso?* (Hansson and Jonsson, 1994).

A ferramenta PRISM é uma ferramenta probabilística de *model checking* capaz de representar e modelar modelos DTMC. O objetivo de realizar o *model checking* é especificado através de expressões PCTL. Esta ferramenta gera modelos na linguagem PRISM. Esta linguagem é baseada no formalismo de modelos reativos e provê abstrações nas cadeias de Markov introduzindo conceitos de alto nível tais como sincronização e modularidade (Kwiatkowska et al., 2007; Alur and Henzinger, 1996).

Para detalhar o relacionamento entre os modelos PRISM/PARAM e a síntese da cadeia de Markov, são introduzidas características importantes da síntese da linguagem PRISM (Alur and Henzinger, 1996; Hahn, 2008; Kwiatkowska et al., 2011, 2007):

1. Cada transição no módulo PRISM tem uma, e somente uma, transição correspondente na cadeia de Markov sintetizada. Essa relação não é invesa.

2. Linguagem PRISM introduz a abstração de módulos PRISM na cadeia de Markov. Esses módulos são executados em paralelo.

3. Módulos PRISM podem sincronizar transições usando rótulos de ação. Quando um módulo alcança uma transição rotulada com o mesmo rótulo de ação de outra transição em outro módulo, o primeiro módulo fica bloqueado esperando pelos outros módulos em execução alcançarem a transição deste rótulo. Neste caso, esses módulos são sincronizados nesse rótulo.

4. Cada transição de módulos diferentes rotulados com a mesma ação são sincronizados em uma transição única na cadeia de Markov com o objetivo de representar a transição de sincronismo.

5. Cada módulo controla internamente os seus estados através da atribuição de valores em um ou mais variáveis de estado. Na cadeia de Markov sincronizada, cada variável de estado tem um estado global do modelo determinado pelos valores de todas as variáveis de todos os modelos. Então, se uma única variável de estado de um único modelo modifica seu valor, então isso representa um novo estado da cadeia de Markov.

4.2.1 Model checking paramétrico probabilístico

Com o *model checking* paramétrico, cada transição é rotulada com parâmetros ao invés de somente constantes. O resultado final desde tipo de *model checking* é parametrizado. Isso permite que sejam feitos diferentes experimentos sem a necessidade de mudar o modelo e fazer um novo *model checking* (Daws, 2005).

A ferramenta PARAM é uma ferramenta para *model checking* paramétrico que virtualmente usa a mesma linguagem de modelagem do PRISM e também tem o mesmo processo de síntese. Assim, a observação para o processo de síntese também é válida para a ferramenta PARAM (Hahn et al., 2010). O PARAM usa como entrada um modelo parametrizado usando extensões na linguagem do PRISM que permite a declaração de parâmetros e expressões PCTL. Essas entradas são usadas para sintetizar a cadeia de Markov e gerar uma fórmula aritmética final como saída. Os termos modelo PRISM e modelo PARA são usados indistintamente neste trabalho.

4.3 Abordagem Proposta

Esta seção apresenta a proposta deste capítulo para estimar confiabilidade em configurações LPS, alavancando o *model checking* paramétrico no domínio de LPS utilizando qualquer tipo de variabilidade. Esta abordagem utiliza como entrada documentação da LPS, tais como modelo de *feature* e *configuration knowledge* (CK) bem como modelos de software comportamental, tais como diagramas UML de sequência, anotados com variabilidade. A Figura 4.1 apresenta uma visão geral do processo de *model checking*.

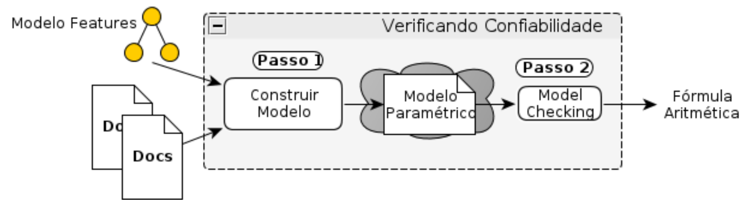


Figura 4.1: Abordagem de Model Checking paramétrico para LPS

No passo 1, esses modelos comportamentais são usados para construção do modelo PARAM que representa a LPS inteira. Este modelo é construído utilizando as seguintes regras:

1. Cada componente de software é mapeado para um módulo PARAM que é mapeado para uma *feature*.
2. Uma mensagem de um componente C para um componente C1 é representado como uma chamada de C para o serviço oferecido por C'. A confiabilidade deste serviço ser executado é a confiabilidade de C1, R'_C, independente do tempo de execução do serviço. Porém, a execução de cada serviço tem a chance de falha associada a execução do componente.
3. Cada *feature* obrigatória do modelo de *feature* é rotulada com o parâmetro correspondente cujo valor varia entre 1 e 0.

4. A variabilidade é tratada através de uma transição de desvio (no nível da modelagem PARAM), que mantem ou desvia daqueles estados que estão relacionados a uma determinada *feature*, de acordo com os seus parâmetros correspondentes (Seção 4.4 para mais detalhes dessa estratégia).

Regras 1 e 2 estabelecem a relação entre modelos comportamentais, o modelo PARAM e o modelo de *features*. Os conceitos de mensagens de comunicação e componentes no diagrama de sequência estão diretamente mapeados nos conceitos de transição e módulo do modelo do PARAM. Os módulos do PARAM estão diretamente mapeados para *features*. Esta estratégia de mapeamento simplifica a validação e automação, restringindo a representação da variabilidade. Em particular, não permite a representação de *features* espalhadas e sobrepostas já que a confiabilidade de um componente é mapeada (Classen et al., 2011; Ghezzi and Sharifloo, 2011b; Rodrigues et al., 2012).

Regras 3 e 4 estão relacionadas a como que a variabilidade é tratada. O parâmetro da regra 3 é usado para definir expressões que são rotuladas como transições de desvio de tal forma que a avaliação, de acordo com a regra 4, é feita conforme presença ou ausência da *feature* no resultado final do *model checking*.

No passo 2 da proposta, o modelo construído no passo 1 é usado como entrada para o PARAM, que por sua vez gera uma fórmula aritmética cujas diferentes avaliações geram valores estimados de confiabilidade diferentes para diferentes configurações. A Figura 4.2 ilustra essa avaliação. Observe que as variáveis relacionadas às *features* são avaliadas com 1 ou 0 (exclusivamente), dependendo da configuração utilizada como entrada.

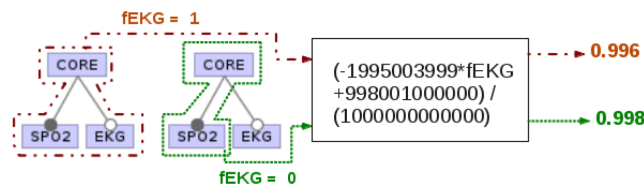


Figura 4.2: Avaliação da fórmula aritmética

Usando essa transição de desvio, é possível pular qualquer parte da cadeia de Markov. Isolando trechos da cadeia é possível selecionar qual parte será alcançada ou não dependendo dos valores das variáveis. Essa alcançabilidade é controlada através da natureza probabilística da cadeia de Markov. Mudando a probabilidade de uma transição para 0, é possível isolar todos os estados necessários para a transição ser alcançada durante o *model checking*.

Qualquer que seja o tipo de variabilidade, esse modelo trata apenas da ausência ou presença da *feature* e seus artefatos relacionados. Restrições de *features* opcionais permitem que a *feature* esteja presente ou não, OR *features* e *features* alternativas permitem que a feature esteja presente ou não dependendo de outras *features*.

De fato, qualquer tipo de *feature* pode ser refinada para *feature* opcional adicionando restrições (Restrições *cross-tree* no modelo de *features*). Figura 4.3 brevemente ilustra a estratégia de refinamento, mostrando que todos os tipos de *features* podem ser refinados para *features* opcionais (Gheyi et al., 2008).

Na abordagem proposta, não é necessário aplicar as restrições do modelo de features durante a fase de modelagem. É preciso inserir as variabilidades de um ponto de variação sem resolver restrições específicas entre elas.

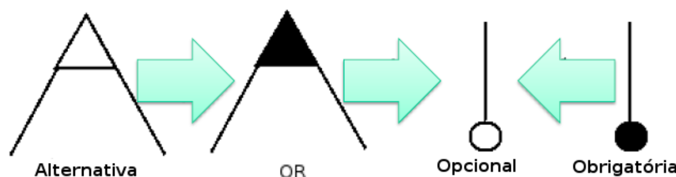


Figura 4.3: Estratégia de refinamento de *features*

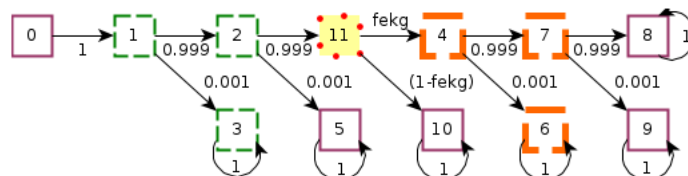


Figura 4.4: Cadeias de Markov paramétricas para LPS

Desta forma, é possível representar qualquer tipo de *feature*, desde que as restrições do produto da LPS são aplicadas pelo engenheiro de aplicação durante a avaliação da fórmula aritmética. Se o tipo de variabilidade de um ponto de variabilidade existente na LPS se modifica, o modelo não precisa se modifica já que as restrições de variabilidade são aplicadas durante a avaliação e não durante a modelagem.

O tratamento de *features* opcionais, o tipo de variabilidade menos restritivo, permite que sejam representados qualquer outro tipo de variabilidade no modelo. Permite também representar de forma simples um mecanismo de desvio que pode pular um conjunto de estados dentro de uma cadeia de Markov.

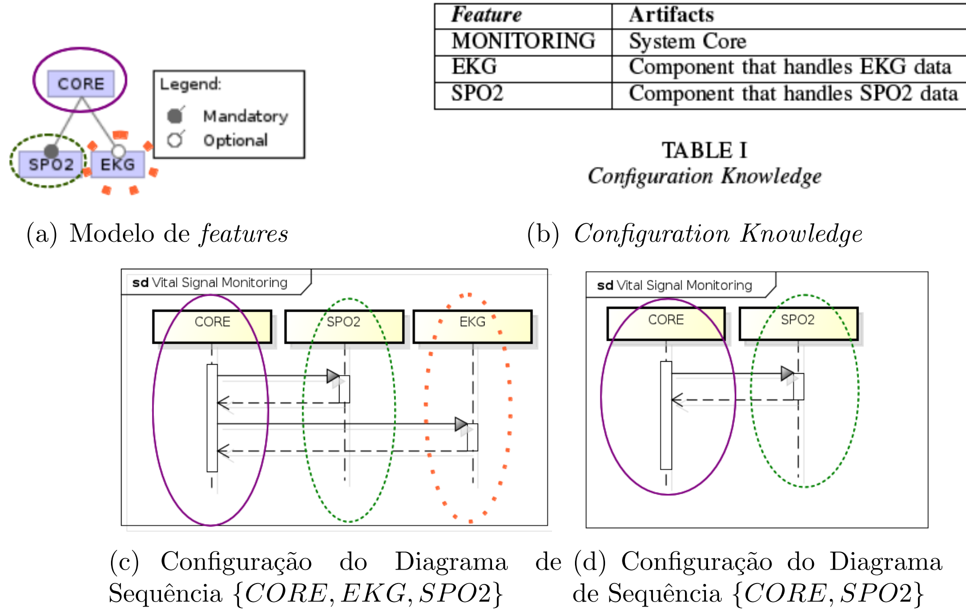
Exemplo de Modelagem

Figura 4.5(a) apresenta um trecho de um modelo de features para um sistema de monitoração de sinais vitais. Este sistema é composto por dois componentes que capturam e tratam os sinais vitais enviados por sensores e provê esses valores ao componente núcleo, que é responsável por identificar situações de risco em relação à saúde do indivíduo monitorado.

Embora esse exemplo seja um trecho pequeno de um modelo de features maior, é o suficiente para ilustrar os conceitos apresentados neste capítulo. Este modelo de features contém duas configurações: $\{CORE, EKG, SPO2\}$ e $\{CORE, SPO2\}$. O exemplo usa um CK simples onde cada componente do sistema é mapeado para uma única *feature*.

Por exemplo, a primeira linha da Fig. 4.5(b) parea a *feature* CORE (núcleo) para o componente núcleo do sistema. É necessário notar que cada *feature* é cercada por suas linhas tracejadas para representar consistentemente as *features* no exemplo.

Figuras 4.5(c) and 4.5(d) representam o diagrama de sequência para cada configuração. Usando esses diagramas de sequência, é construído dois modelos PARAM não parametrizados (seguindo as regras apresentadas nessa seção). Esses modelos PARAM sintetizam as cadeias de Markov apresentadas nas Figuras 4.6(a) e 4.6(b), respectivamente. Note que uma configuração tem parte do seu modelo comportamental (diagrama de sequência) similar a outra configuração. As cadeias de Markov correspondentes também possuem similaridades que podem ser mapeadas por uma numeração dos estados. Cada estado é unicamente numerado para diferenciar estados não relacionados no modelo. Por causa



Feature	**Artifacts**
MONITORING	System Core
EKG	Component that handles EKG data
SPO2	Component that handles SPO2 data

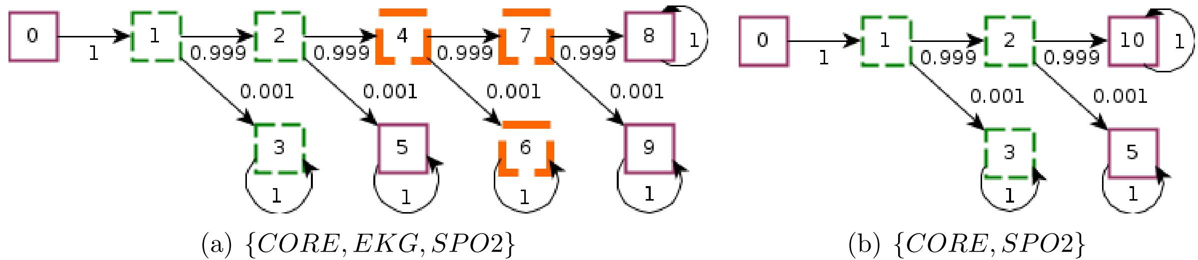
TABLE I
Configuration Knowledge

(a) Modelo de *features* (b) *Configuration Knowledge*

(c) Configuração do Diagrama de Sequência $\{CORE, EKG, SPO2\}$ (d) Configuração do Diagrama de Sequência $\{CORE, SPO2\}$

Figura 4.5: Documentação da LPS

dessas particularidades no modelo de síntese do PARAM (Descrito na Seção 4.2), para alguns estados não existe nenhum mapeamento n-para-n entre os módulos do modelo PARAM e estados da cadeia de Markov.



(a) $\{CORE, EKG, SPO2\}$ (b) $\{CORE, SPO2\}$

Figura 4.6: Cadeias de Markov das configurações da LPS

Usando a abordagem proposta, a cadeia de Markov correspondente ao modelo PARAM é apresentada na Figura 4.4. A cadeia de Markov parametrizada é capaz de representar os modelos presentes nas Figuras 4.6(a) e 4.6(b).

O modelo da Figura 4.4 explora as comunalidades entre esses dois modelos reutilizando os estados $0, 1, 2, 3, 5$, adicionando somente estados do modelo da Figura 4.6(a) e estados de controle 11 para tratar a variabilidade.

Essa abordagem evita a replicação de estados por meio da combinação de diversos modelos em um único modelo. A cadeia de Markov apresentada na Figura 4.4 é parametrizada pelos parâmetros $fEKG$ que podem ser utilizados para pular partes da cadeia com o objetivo de calcular a confiabilidade de uma configuração específica. O estado numerado com 11 é o estado de controle introduzido para decidir se uma parte da cadeia é isolada ou não. Esta cadeia de Markov resulta em uma fórmula aritmética que pode ser avaliada, conforme ilustrado na Figura 4.2.

A abordagem proposta está baseada na natureza probabilística da cadeia de Markov. A variável da *feature* EKG ($fEKG$) é usada para selecionar ou não a *feature* correspondente. Para tal, é necessário atribuir valores 1 ou 0, respectivamente, para a variável. Nas cadeias de Markov, essa avaliação possui o significado de que uma transição tem 100% ou 0%, respectivamente, de ser executada. Nas próximas seções, é discutido este mecanismo e apresentada uma prova de validade (soundness) dessa abordagem.

4.4 Prova de Validada (*Soundness*)

Este prova consiste do detalhamento de uma cadeia de Markov sintetizada pelo modelo PARAM que possa representar cadeias de Markov para cada configuração diferente, dependendo da avaliação dos parâmetros para seleção de *features*.

O exemplo da Seção 4.3 apresenta duas cadeias de Markov não parametrizadas (Figura 4.6(a) e 4.6(b)) e uma cadeia de Markov parametrizada capaz de representa as outras duas de acordo com diferentes valores do parâmetro $fEKG$ (Figura 4.4).

Conforme descrito na Seção 4.2, as cadeias de Markov podem ser utilizadas para realizar o *model checking* de propriedades a partir da propriedade de alcançabilidade. Utilizando esse conceito da teoria dos grafos, é possível demonstrar que todos os caminhos da cadeia de Markov de uma configuração simples possuem caminhos equivalentes no modelo parametrizado quando as valorações para as *features* selecionadas estão de acordo com o representado na configuração.

Esta equivalência é definida em termos da computação da probabilidade dos caminhos (multiplicação dos rótulos das transições). Note que todos os caminhos na cadeia parametrizada podem ser ligeiramente diferentes do caminho da cadeia de Markov resultante. Será demonstrado que essa diferença está limitada aos estados de controle e transição de decisão e que esses estados e transições não modificam o resultado computador. Essas premissas garantem que o valor de confiabilidade é o mesmo para um modelo com uma única configuração.

4.4.1 Demonstração

Esta prova é composta por dois passos: 1) é definido um subconjunto de estados na cadeia de Markov e definido como esses estados de controle são corretamente trocados quando as features estão presentes ou não; 2) é demonstrado i) como a cadeia de Markov parametrizada da LPS é avaliada de acordo com a configuração da LPS e ii) como a cadeia de Markov a cadeia resultante possui caminhos equivalentes.

O primeiro passo apoia o segundo mostrando que para uma dada configuração, somente os estados da configuração e os estados de controle estão acessíveis.

Uma cadeia de Markov é definida como um grafo (Definição 1) onde as arestas são rotuladas com expressões definidas nos conjunto dos números reais $\mathbb{R}$ (Definição 2). Estas expressões representam as probabilidades de transições nas cadeias Markov.

Definição 1

Uma cadeia de Markov sintetizada a partir de um modelo PARAM é um dígrafo (Grafo direcionado) denotado por $G = (V, E)$ onde V é o conjunto de vértices, E é o conjunto de arestas e $(u, v) \in E$ denota uma aresta direcionada que parte de

um vértice u e chega no vértice v, $u, v \in V$. A aresta (u, v) é dita *incidente* em v (Kwiatkowska et al., 2011; Grinstead and Snell, 2006).

Definição 2 Seja $p : E \rightarrow \mathbb{R}$ a função cuja a aresta do gráfico é rotulada com uma expressão aritmética representando a probabilidade de uma cadeia de Markov transitar de um vértice u para v.

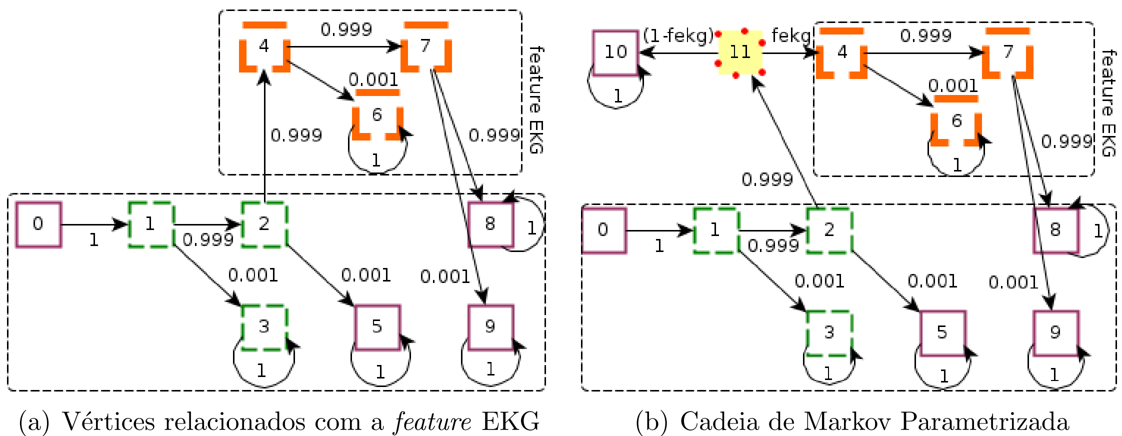
Valores válidos são definidos no intervalo fechado de $[0, 1]$ onde 0 representa 0% de probabilidade e 1 representa 100% de probabilidade de executar uma transição.

De agora em diante, a cadeia de Markov é considerada um dígrafo e os termos *cadeia de Markov*, *grafo* e *dígrafo* podem ser usados de forma indistinta. Da mesma forma, os termos *transições* e *arestas*, bem como *estados* e *vértices*. A cadeia de Markov não possui estados desconectados, ou seja, não existe estado que não tenha uma transição de saída ou de entrada no estado. Lema 1 define essa propriedade.

Lema 1. *A cadeia de Markov sintetizada de um modelo PARAM é um dígrafo conectado (Bondy and Murty, 2008; Kwiatkowska et al., 2011; Grinstead and Snell, 2006).*

Por causa da regra 1 da Seção 4.3, todos os estados associados a um determinada *feature* F estão confinados em um módulo PRISM. Essa propriedade permite rastrear todos os estados associados a uma determinada *feature* de um modelo de *features* para uma cadeia de Markov.

Definição 3 define este subconjunto de estados relacionados a uma feature específica. Figura 4.7(a) exemplifica o subconjunto S com os estados e transições relacionados à feature EKG (Estado laranja, delimitado por linhas grossas tracejadas).



(a) Vértices relacionados com a *feature* EKG (b) Cadeia de Markov Parametrizada

Figura 4.7: Tratamento de variabilidade

Definição 3 Seja S um subconjunto de vértices $S \subseteq V$ tal qual se s é um vértice (estado) associado com uma feature opcional F então $s \in S$.

Este subconjunto induz um grafo que contém todos os estados e transições relacionados com a execução de um componente associado com uma determinada feature, representando S. Um subconjunto complementar $(V \setminus S)$ também induz o grafo que, por definição, é disjunto de S (Definição 4). Estes subgrafos induzidos são usados para descrever como os estados relacionados a feature F (subconjunto S) pode ser desviado durante o *model checking*. Neste contexto, o corte (Definição 5) possui um papel importante desde que define o subconjunto S, subconjunto de arestas que entram e saem do vértice, essencial para mostrar que os estados podem ser desviados. Figura 4.8(a) ilustra as definições 3, 4 e 5.

Definição 4 Denota-se por $G[S]$ o subgrafo de G induzidor pelos vértices de S e $G[V \setminus S]$, o subgrafo induzido pelo subconjunto de vértices $V \setminus S$.

Definição 5 Denota-se por $\partial^-(S)$ e $\partial^+(S)$, os subconjunto de arestas que entra e saem de V associados com S, respectivamente.

$\partial(S) = \partial^-(S) \cup \partial^+(S)$ é o *corte* V associado com S (Bondy and Murty, 2008).

Lema 2. *O subgrafo induzido, $G[S]$, é um gráfio direcionado.*

Demonstração. A partir da Definição 1, $G(V, E)$ é um grafo direcionado. Desde que o grafo induzido contem todas as arestas direcionadas entre os vértices de $S \subseteq V$, o subgrafo $G[S]$ é também direcionado. □

Todas as arestas do subconjunto de arestas incidentes de S chegam ao mesmo vértice (Lema 3). Esta propriedade da cadeia de Markov obtida na abordagem de modelagem proposta permite que os estados de S sejam facilmente desviadas, desde que um único ponto de entrada inicia as transições de uma *feature* F.

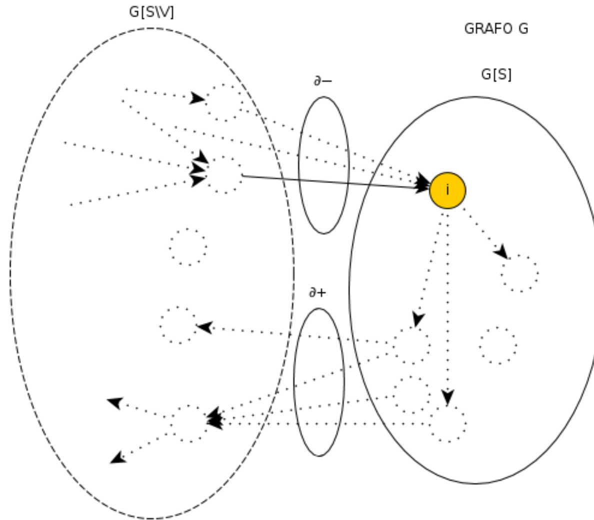
Definição 6 Seja $av : E \rightarrow V$ uma função tal que $av((u, v)) = v$. Esta função retorna a aresta que incide em um nó.

Lema 3. $\exists i | \forall (u, v) \in \partial^-(S)$, $av((u, v)) = i$

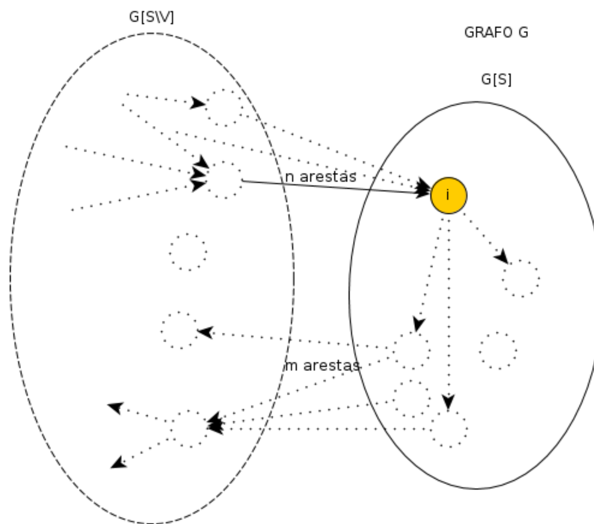
Demonstração. Este resultado vem da Regra 1 da Seção 4.3 e item 3 e 4 da Seção 4.2. □

Todas as arestas incidentes para $G[S]$ ($\partial^-(S)$) chegam no mesmo vértice i. Esta é uma importante propriedade da proposta desde que todos os caminhos originados de um vértice em $V \setminus S$ que visitam algum vértice em S chegará eventualmente a visitar o vértice i. Figura 4.8(b) ilustra esse comportamento. Note que na Figura 4.7, o vértice incidente correspondente ao vértice i da definição é o vértice rotulado com o número 4.

Definição 7 distingue o grafo do modelo paramétrico G do grafo de um modelo para uma única configuração G_c e as suas correspondentes definições de corte, corte de entrada e corte de saída, conjunto de vértices e conjunto de arestas. Estas definições são diferenciadas por um subscrito c e suas definições. Esta diferenciação é importante para



(a) $G[S]$, $G[V \setminus S]$, $\partial^+(S)$ e $\partial^-(S)$



(b) Grafo G

Figura 4.8: $G(V, E)$

definir o formato de um grafo parametrizado e a relação com os grafo não parametrizado conforme Definição 9. Esta definição mostra como um grafo de uma configuração da LPS é modificado para tratar variabilidade.

Para tornar as *features* de F opcional, é introduzido um estado de controle f na cadeia de Markov de tal forma que possa seletivamente desviar os estados relacionados a esta *feature* sem modificar o valor de probabilidade da execução de um caminho existente. Este estado de controle é introduzido entre $G_c[S]$ e $G_c[V_c \setminus S]$ de tal forma que todas as arestas que incidem em i passam a incidir no estado de controle f e f passa a ter duas aresta de saída: uma incidente em $G_c[S]$ através do vértice i e uma outra aresta incidente em algum dos vértices representando a configuração possível da LPS sem a *feature* F. Figura 4.9(a) ilustra a introdução do vértice f e suas correspondentes arestas. A função desses vértices pode ser identificada na Figura 4.7(b), onde f é um vértice numerado com 11 e suas arestas de saída apontam para um estado relacionado à *feature* EKG e para outro

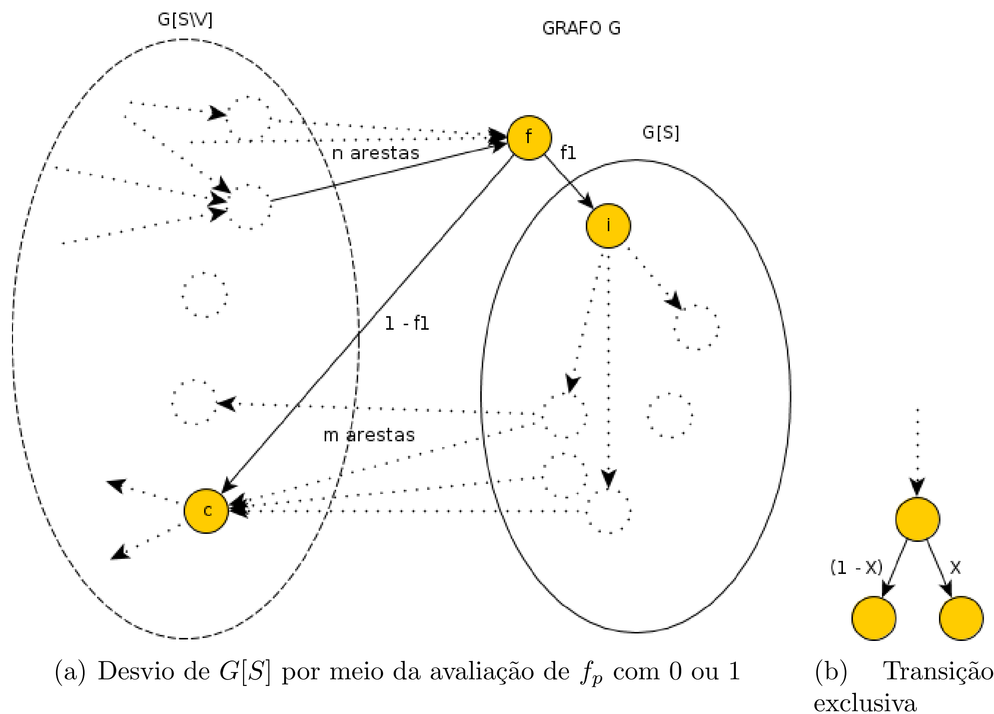
estado representando a execução do sistema sem a *feature*.

As transições que saem do vértice de controle f são rotuladas com expressões que envolvem a seleção do parâmetro da *feature* (Definição 8) de tal forma que a avaliação desse parâmetro desvia os vértices de S.

Definição 7 Denota-se por $G_c(V_c, E_c)$ o grafo representando a cadeia de Markov de uma configuração única sem a variabilidade, $\partial_c^-(S)$ e $\partial_c^+(S)$ os cortes de entrada e saída de S com relação s $V_c \setminus S$, respectivamente.

Definição 8 Seja f_p um parâmetro para seleção de *feature* associado com a *feature* F, então $f_p \in \{0, 1\}$.

Definição 9 Seja G_c um grafo de uma configuração da LPS, $S \subseteq V_c$ o subconjunto de vértices relacionados com uma dada feature F e $i \in S$ um vértice definido de acordo com o Lema 3, então G é o grafo G_c parametrizado com o vértice f tal qual $\forall(u, v) \in \partial_c^-(S)$, $av((u, v)) = f$ e $\partial^-(S) = \{(f, i)\}$ e f tem duas arestas que saem do vértice (f, i) e (f, c) onde c é um vértice arbitrário $c \notin S$, $p(f, i) = f_p$, $p(f, c) = 1 - f_p$



(a) Desvio de $G[S]$ por meio da avaliação de f_p com 0 ou 1 (b) Transição exclusiva

Figura 4.9: Tratamento de variabilidade

As duas arestas que saem do vértice f são rotulados com expressões que podem seletivamente fazer com que a cadeia de Markov isole os vértices de S. Isto é possível utilizando as expressões que envolvem o parâmetro f_p da *feature* F.

Rotulando uma aresta com f_p e outra com $(1 - f_p)$ e avaliando f_p com 0 ou 1, é possível trocar as transições da cadeia de Markov, controlando o desvio dos estados relacionados a *features* F (Lema 4).

Na Figura 4.7(b), o estado de controle numerado com 11 tem duas transições que saem do vértice com expressões associadas ao parâmetro de seleção da *feature* ekg. Se $ekg = 1$, os estados relacionados com a *feature* EKG serão acessíveis via a aresta com 100% de chance de sucesso, senão ($ekg = 0$), esses estados serão ignorados dado que a chance de transição é 0% e a cadeia de Markov passará para a outra transição de saída de 11, chegando no estado da cadeia representando o sistema sem a *feature* EKG.

Lema 4. *Sejam* (s, r) *e* $(s, t) \in E$ *duas arestas e* f_p *um parâmetro para seleção da feature tal que* $p(s, r) = f_p$*,* $p(s, t) = (1 - f_p)$*, e* $\nexists(s, v)$ *tal que* $v \neq t$*,* $v \neq r$ *(Figura 4.9(b)). Então,* (s, r) *e* (s, t) *são exclusivas, i.e., uma vez que avalia-se* f_p*, a cadeia de Markov sempre alcança uma dessas arestas.*

Demonstração. $\forall(u, v)$, $\sum p(u, v) = 1$ (Kwiatkowska et al., 2011; Grinstead and Snell, 2006). Então, $p(s, r) + p(s, t) = 1$, então $p(s, r) = 1 - p(s, t)$. Por Definição 2 $p(s, r) = X(X \in [0, 1] \in \mathbb{N})$, então $p(s, t) = Y = (1 - X)$ □

Esses lemas e definição até então, demonstram como que a abordagem proposta identifica e isola trechos da cadeia de Markov relacionados a uma *feature*. Os próximos lemas demonstram que, apesar a adição de novos vértices, cada possível configuração e cada caminho da cadeia de Markov possui um caminho equivalente na cadeia de Markov parametrizada.

Esta equivalência é definida em termos da probabilidade da execução de cada caminho. A probabilidade de cada caminho é calculada pela multiplicação consecutiva das expressões que rotulam cada aresta do caminho. Para calcular a probabilidade a alcançar um determinado estado a partir de outro na cadeia de Markov, é feita uma soma das probabilidade de todos os caminhos entre esses estados. São apresentadas algumas definições relacionadas à caminho (Definições 10, 11) e alguns resultados (Lemas 5, 6) que sã consequência de resultados anteriores para introduzir um resultado final.

Definição 10 Denota-se por $u \rightsquigarrow v$ qualquer caminho que se inicie a partir de um vértice u e finalize no vértice v. Note que esse é um caminho direcionado.

Definição 11 O conjunto ordenado de todas as arestas de um $u \rightsquigarrow v$ é denotado por $E(u \rightsquigarrow v)$.

Lema 5. *Considere todos* $u \rightsquigarrow v$ *caminhos de* $G(V, E)$ *tais que* $u \in V \setminus S$ *e* $S \cap E(u \rightsquigarrow v) \neq \emptyset$*, então* $i \in E(u \rightsquigarrow v)$*.*

Demonstração. Suponha que existe um $u \rightsquigarrow v$ que não tenha visitado i, então existe uma aresta (a, b) tal qual $a \in V \setminus S$ e $b \in S$ e $b \neq i$, que contradiz a unicidade do vértice incidente i de S (Lema 3) , então esse caminho não existe. □

Lema 6. *Considere todos os $i \rightsquigarrow s$ caminhos de $G(V, E)$ tais que $s \in S$ e $E(i \rightsquigarrow s) \subseteq S$, então $E(i \rightsquigarrow s)$ definido no $G = E(i \rightsquigarrow s)$ definido G_c.*

Demonstração. Note que S é definido em G, G_c e não é modificado por Definição 9. □

O teorema final (Teorema 1) afirma que para uma determinada configuração, o *model check* na cadeia de Markov parametrizada e em uma cadeia de Markov não parametrizada, para o mesmo PCTL e configuração da LPS, o resultado do valor de confiabilidade é o mesmo. Para enunciar este Teorema, é definida uma função que representa o cálculo do *model check* de uma dada cadeia de Markov e expressão PCTL (Definição 14). Expressões PCTL selecionam caminhos de interesse na execução do *model checking* (Definição 13). O resultado final do model checking é obtido pelo cálculo de probabilidade da seleção de cada caminho, conforme definido pela função *pp* (Definição 12).

Definição 12 Seja $pp : P \rightarrow \mathbb{R}$ onde P é um conjunto de caminhos definidos na cadeia de Markov associada ser uma função que calcula a probabilidade de um caminho. Esta probabilidade é calculada pelo produto de todas os valores de probabilidade que rotulam as arestas presentes neste caminho: $pp(u \rightsquigarrow v) = \prod_{(x,y) \in E(u \rightsquigarrow v)} p(x, y)$

Definição 13 Seja $sp : M, PCTL \rightarrow 2^P$ onde M é um conjunto de cadeia de Markov, $PCTL$ um conjunto de expressões PCTL e 2^P o conjunto de todos os caminhos definidos na cadeia de Markov associada ao grafo ser uma função que retorna todos os caminhos selecionados pela cadeia de Markov associada ao grafo por uma expressão PCTL.

Definição 14 Seja $mc : M, PCTL \rightarrow \mathbb{R}$ onde M é um conjunto de cadeias de Markov e $PCTL$ o conjunto de expressões PCTL se uma função que calcula a probabilidade do valor de uma cadeia de Markov associada com uma dada expressão PCTL: $mc(m, e) = \sum\limits_{p \in sp(m,e)} pp(p)$

Para provar esse teorema, mostra-se que cada caminho $u \rightsquigarrow v$ de uma configuração da cadeia de Markov tem um caminho equivalente $u \rightsquigarrow v$ na cadeia de Markov parametrizada com os parâmetros avaliados pela seleção de uma configuração da LPS.

Definição 15 confirma essa relação de equivalência. Dois caminhos são ditos equivalentes se $E(u \rightsquigarrow v)$ diferem-se somente pela introdução dos estados de controle e se suas probabilidades são iguais. Esta propriedade garante que qualquer *model checking* em uma cadeia de Markov parametrizada resulta no mesmo valor de uma cadeia de Markov não parametrizada.

Definição 15 Seja p_1 e p_2 serem dois caminhos, então p_1 é dito equivalente para p_2 se $pp(p_1) = pp(p_2)$ e $E(p_2) \setminus C) = E(p_1$ tais quais C é um conjunto de estados de controle conforme definido em Definição 9.

Teorema 1. *Seja* FM *um conjunto de configurações da LPS,* M_p *se um conjunto de modelos de cadeia de Markov parametrizadas conforme definição da Seção 4.3 e* M *um conjunto de modelos de cadeias de Markov não parametrizadas construídas aplicando as Regras 1 e 2 da Seção 4.3 para uma dada configuração* $c \in FM$*.*

Seja $npm : FM \rightarrow M$ *uma função de mapeamento entre as configurações e seus modelos parametrizados correspondentes.*

Seja $ck : FM, M_p \rightarrow M$ *uma função que avalia todos os parâmetros de seleção de features de* $m_p \in M_p$ *de acordo com Definições 8 e 9 e retorna o modelo não parametrizado representando* $c \in FM$*.*

Então $mc(npm(c), e) = mc(ck(c, m_p), e)$*,* $\forall e | e \in PCTL$ *é uma expressão definida nos estados* $npm(c)$*.*

Demonstração. Por Definição 9, $ck(c, m_p)$ possui todos os estados de $npm(c)$. Isso garante que todas as expressões PCTL definidas em $npm(c)$ podem ser definidas em $pm(c)$ (Limitadas a expressões que não consideram o tempo na execução, dado que transições adicionais modificam a chance da execução esperada em um número específico de passos).

Por Definição 9 e Lema 4, caminhos selecionados podem diferenciar-se somente pelos estados de controle. Caminhos que não pertencem à configuração selecionada têm o valor igual pp a 0. Então, somente caminhos válidos no que diz respeito à configuração da LPS, podem modificar os valores de mc.

É suficiente mostrar que todos os caminhos em $sp(npm(c), e)$ têm um caminho equivalente em $sp(ck(c, m_p), e)$ em termos de relação da Definição 15.

Seja $G = ck(c, m_p)$ e $G_c = npm(c)$. Existe quatro tipos de caminhos no grafo da cadeia de Markov parametrizada que precisam ser avaliados:

1. $u \rightsquigarrow v$ tal qual $u \in G_c[V_c \setminus S]$ e $v \in G_c[V_c \setminus S]$:

   Não existe caminho G_c que visite o vértice f já que não existe este grafo. Então, cada caminho de $G_c[V_c \setminus S]$ tem um caminho equivalente em $G[V \setminus S]$ pois nenhum vértice ou aresta foi removido de $G_c[V_c \setminus S]$ para isolar S em $G[V \setminus S]$.

2. $u \rightsquigarrow v$ tal qual $u \in G_c[V_c \setminus S]$ e $v \in G_c[S]$:

   Pelos Lemas 3, 4 e 5, todo caminho que visita i em G_c passara por f em G, mas $p(f, i) = 1$ quanto a seleção de features f_p é avaliada para representar uma configuração específica. Então, esta aresta adicionar nesses caminhos não modificam a probabilidade desses caminhos (o produtório das expressões que rotulam a aresta).

3. $u \rightsquigarrow v$ tal qual $u \in G_c[S]$ e $v \in G_c[V_c \setminus S]$:

   Pelo lema 6 e item 1 desta prova, os caminhos que iniciam-se no vértice de S e finalizam-se no vértice de $G[V \setminus S]$ são iguais aos caminhos em G_c, já que nenhuma mudança nas arestas ou vértices foram feitas nesse caminho.

4. $u \rightsquigarrow v$ tal qual $u \in G_c[S]$ e $v \in G_c[S]$:

   Resultados dos Lemas 3 e 6.

□

4.5 Conclusão

Este capítulo propõe uma abordagem para lidar com o importante problema de calcular propriedades não funcionais da LPS. Em particular, trata da propriedade de confiabilidade.

A abordagem proposta está alinhada com os trabalhos relacionados no sentido de que usa um único modelo para representar a LPS inteira.

Nossa abordagem supera as limitações dos trabalhos atuais para modelar LPS utilizando ferramentas paramétricas para *model checking* propondo uma forma para tratar *features* opcionais. Nós também discutimos a importância das *features* opcionais como um caso mais geral de variabilidade, já que nas abordagens de model checking paramétrica, as restrições entre *features* são obrigatórias no tempo de avaliação.

Apresenta-se também uma prova de validação (*sound*) da abordagem proposta para tratar *features* opcionais e a representação correta para a documentação, temas não tratadas por trabalhos similares atuais.

Capítulo 5

Model Checking Composicional em Linhas de Produto de Software

Model Checking paramétrico é uma técnica que permite a avaliação tardia de parâmetros em um modelo. Essa técnica produz como saída uma fórmula aritmética parametrizada com parâmetros definidos no modelo. Essa flexibilidade pode ser utilizada para lidar com variabilidades de LPSs diretamente nos modelos evitando o trabalho inviável de construir um modelo diferente para cada diferente produto de um LPS como discutido no capítulo 3 e em trabalhos relacionados (Ghezzi and Sharifloo, 2011b).

Esse formato de saída permite uma ampla gama de alternativas no que diz respeito ao tempo de avaliação da fórmula como por exemplo tempo de projeto ou execução. Apesar das propostas de modelagem apresentadas focarem no uso de parâmetros para determinar a configuração da LPS, outros parâmetros podem ser adicionados ao modelo com diferentes significados e diferentes intervalos de valores válidos. Uma vez que todas as possíveis configurações serem conhecidas em tempo de execução, não há sentido em avaliar a fórmula para diferentes configurações em tempo de execução. Entretanto, pode haver outros parâmetros cujo valor muda em tempo de execução, por exemplo uma aplicação pode depender de um componente cuja confiabilidade muda em tempo de execução devido a políticas de economia de energia requerendo que seu valor correspondente no modelo seja uma parâmetro ao invés de uma constante. Assim, em alguns cenários a avaliação da fórmula em tempo de execução precisar ser viável e isso está diretamente relacionada com seu tamanho (ver Capítulo 3).

Ao longo desse trabalho foi utilizada a ferramenta de *model checking* paramétrico PARAM para a obtenção da fórmula parametrizada. A fórmula final obtida por essa ferramenta é apresentada em sua forma completamente expandida, ou seja, a propriedade distributiva não pode ser mais aplicada. Esse formato de fórmula é uma escolha de implementação da ferramenta PARAM e , como discutido anteriormente (ver Seção 3.5), pode levar a fórmulas grandes em alguns modelos.

Avaliar uma fórmula em tempo de execução traz à tona algumas questões práticas relacionadas com seu tamanho (ver Seção 3.5.2). Mesmo fórmula com milhares de operandos podem ser avaliadas em milissegundos em tempo de execução em computadores modernos. Entretanto, o mesmo não é verdade quando se trata de dispositivos móveis com recursos limitados.

Esse capítulo trata esse problema apresentado uma abordagem composicional para *model checking* paramétrico capaz de produzir fórmulas parcialmente fatoradas. A fatoração polinomial é uma alternativa para reduzir o esforço de avaliação e *parsing* da fórmula final por meio da redução de redundâncias observadas na fórmula completamente expandida e, assim, reduzindo o tamanho da fórmula.

A abordagem proposta neste capítulo divide o modelo em partes independentes de tal forma que cada parte pode ser modelada e verificada separadamente produzindo fórmulas independentes. A seguir, essas fórmulas são compostas para gerar uma fatoração parcial da fórmula correspondente à obtida pelo *model checking* de toda a LPS. Essa abordagem simplifica o *model checking* já que é feita a verificação de modelos menores separadamente ao invés de um único grande modelo e o esforço de recombinar essas fórmulas é tão simples quanto substituição de texto.

Inicialmente, será apresentada a abordagem de modelagem (Seção 5.1), em seguida será discutida a intuição por trás da redução da fórmula por meio de fatoração (Seção 5.2) e por fim serão comparados os tamanhos das fórmulas obtidas por meio da abordagem composicional e da não composicional (Seção 5.3).

5.1 *Model Checking* Paramétrico Composicional

Essa abordagem de modelagem utiliza como entrada diagramas de sequência e de atividades UML. Esses diagramas precisam ser construídos de tal forma que o diagrama de atividades represente uma visão mais ampla de um cenário e cada atividade é detalhada por diagramas de sequência. Assim, há dois níveis de abstração para um dado cenário. No primeiro nível, que provê uma visão mais ampla, é chamado nível de atividades; o segundo nível, que detalha cada atividade, é chamado de nível de componente uma vez que os diagramas de sequência descrevem interações entre componentes. A Fig. 5.1 apresenta uma diagrama de atividades com uma de suas atividades detalhada por diversos diagramas de sequência.

É chamado de *caminho-seta* um caminho que conecta uma atividade a outra, possivelmente passando por algum nó de decisão (forma de diamante). Por exemplo, a terceira atividade da Fig. 5.1 apresenta dois caminhos-sera de saída: um chega à quarta atividade e outros chega ao estado final. Cada caminho-seta de saída representa um estado final diferente no modelo de nível de componente relacionada à atividade.

No primeiro nível, cada caminho-seta representa uma execução atômica independente com uma fórmula de confiabilidade associada. Cada execução é considerada atômica uma vez que esta não pode ser mais detalhada nesse nível (nível de atividades). O valor de confiabilidade de cada atividade é calculado utilizado a confiabilidade dos componentes de acordo com seus diagramas de sequência subjacentes. No nível de componentes, diagramas de sequência relacionados com cada atividade são utilizados para construir um modelo como descrito nos Capítulos 3 e 4. Cada diferente estado final de sucesso de um dado modelo a nível de componente é considerado uma execução diferente da atividade correspondente.

A Fig. 5.2 ilustra o processo de obtenção da fórmula fatorada dessa abordagem. Esse processo compreende os seguintes passos:

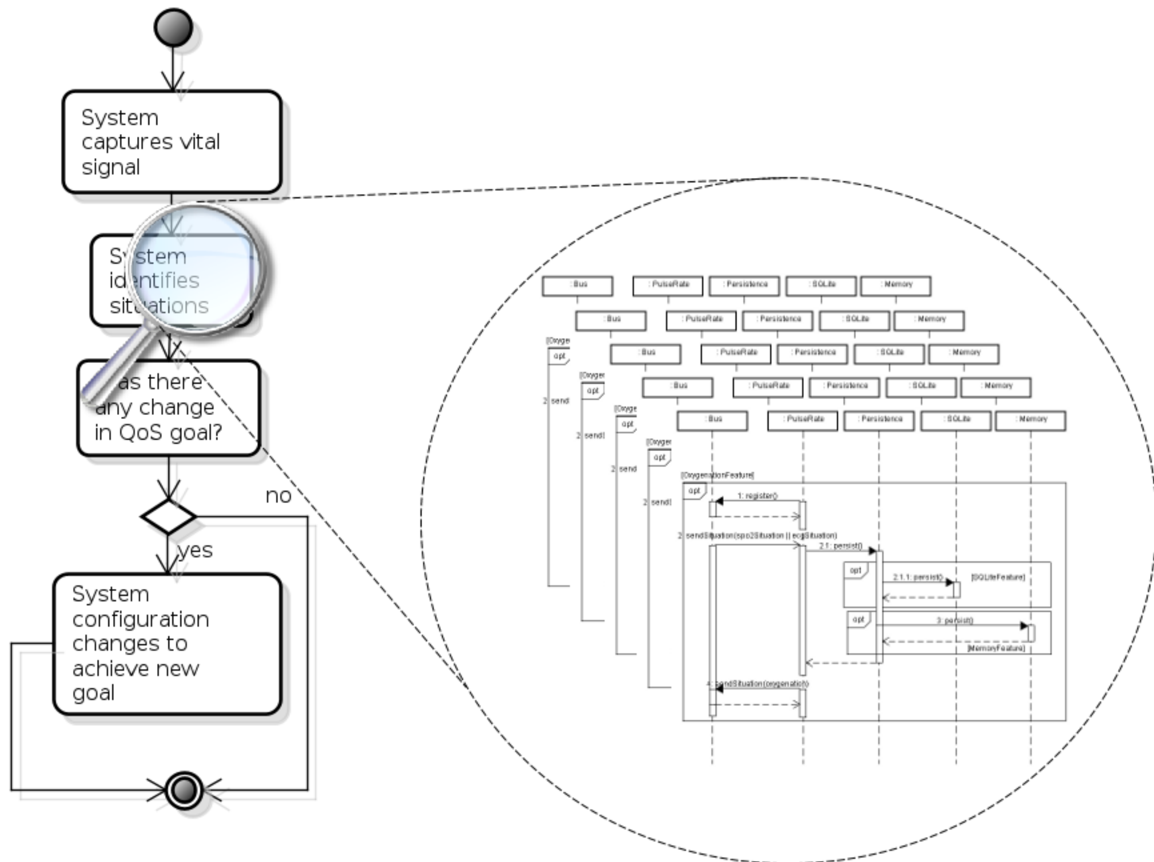


Figura 5.1: Diagramas de Sequência Detalhando uma Atividade

1. construir o modelo PARAM no nível de componente correspondente à atividade. Esse passo utiliza como entrada o modelo de *features*, diagramas UML, o CK e qualquer documentação adicional necessária para interpretar os diagramas;

2. construir o modelo PARAM no nível de atividades. Esse passo utiliza como entrada as mesmas entradas do nível de componentes, exceto a documentação relacionada a LPS.

3. fazer o *model checking* de cada modelo no nível de componentes e obter suas fórmulas correspondentes.

4. fazer o *model checking* do modelo no nível de atividades para obter a fórmula completamente parametrizada representando a fatoração;

5. as fórmulas obtidas no terceiro passo são combinadas da forma como a fórmula definida no quarto passo define resultando na fórmula fatorada.

Note que o passo 2 não utiliza como entrada a documentação relacionada com variabilidade, uma vez que essa abordagem não lida com variabilidades ano nível de diagrama de atividades.

O modelo no nível de atividades referido no passo 2 é construído de acordo com as seguintes regras:

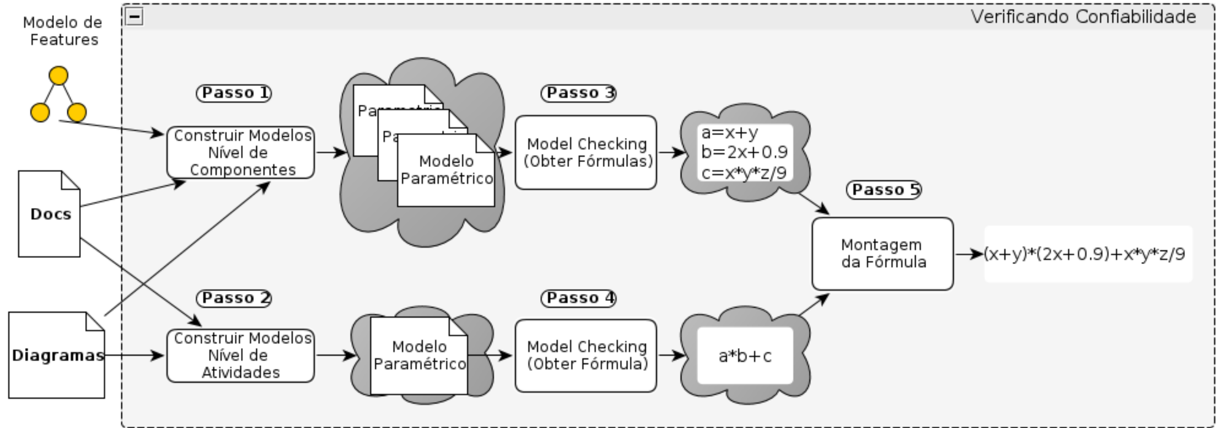


Figura 5.2: Processo de *model checking* paramétrico composicional

1. cada caminho-seta possui om parâmetro correspondente v, tal que $v \in V$ onde V is é o conjunto de parâmetros de caminhos-seta representando sua probabilidade de sucesso associada.

2. O diagrama de atividades é modelo como um único modelo PARAM.

3. Cada atividade em $a \in A$ tal que A é o conjunto de atividades, é modelado como um estado único.

4. Cada caminho-seta $(a_s, a_d) \in A \times A$, onde a_s é a atividade origem e a_d é a atividade destino do caminho seta, é mapeado para uma transição no modelo PARAM deixando o estado associado com a atividade a_s e chegando no estado associado com a atividade a_d. Cada transição PARAM é rotulada com um parâmetro diferente v dado que $L : A \times A \to V$ é um a função de rotulamento tal que $L((a_s, a_d)) = v$.

5. Para cada estado do modelo PARAM associado com uma atividade a_s, é adicionada uma transição representando a chance de falha associada com cada atividade dada por $(1 - (\sum_{(a_s,a_d) \in A \times A} L((a_s, a_d))))$.

Cada atividade compreende uma ou mais execuções atômicas, assim não é necessário detalhar sua execução em um módulo PARAM separado com apenas um estado (Regras 2 e 3). Cada diferente caminho-set no nível de atividade leva a uma próxima atividade diferente. Portanto, um estado no modelo no nível de atividades tem uma transição de saída para cada diferente estado final de seu correspondente modelo no nível de componentes e cada uma dessas transições é rotulada com um parâmetro diferente (Regras 1, 4). Para cada caminho-seta há uma fórmula correspondente obtida com os diagramas de sequência associados com a atividade origem do caminho-seta que será utilizada como substituição para seu parâmetro correspondente no passo 5 do processo (ver Fig. 5.2). Cada atividade possui uma chance de falha associada representando a chance de falha agregada para todos os caminho-seta de uma atividade (Regra 5).

Por exemplo, Fig. 5.3 e Listagem 5.1 apresentam um diagrama de atividades e sue correspondente modelo ao nível de atividades. Cada estado é rotulado com uma ação para melhor ilustrar a correspondência entre o modelo e o diagrama.

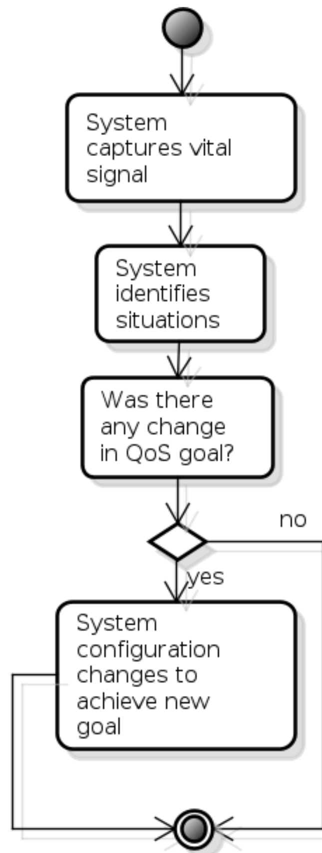


Figura 5.3: Diagrama de Atividades

Listagem 5.1: Modelo PARAM no Nível de Atividades

```
dtmc

param double capture;
param double situation;
param double qosgoal1;
param double qosgoal2;
param double reconfiguration;

module VitalSignalMonitoring

        s : [0..5] init 1;

        [FAIL] s = 0 -> (s'=0);

        [SYSTEM_CAPTURE_VITAL_SIGNAL] s = 1 -> capture : (s'=2) +
                                   (1-capture) : (s'=0);
        [SYSTEM_IDENTIFIES_SITUATION] s = 2 -> situation : (s'=3) +
                                   (1-situation) : (s'=0);
        [CHANGE_QOS_GOAL] s = 3 -> qosgoal1 : (s'=4) +
                           qosgoal2 : (s'=5) +
                  (1-(qosgoal1 + qosgoal2)) : (s'=0);
        [SYSTEM_RECONFIGURE] s = 4 -> reconfiguration : (s'=5) +
                              (1-reconfiguration) : (s'=0);
        [END] s = 5 -> (s'=5);

endmodule
```

5.2 Argumento

A premissa por trás da abordagem composicional apresentada é que a fatoração leva a fórmulas simples por meio da redução de redundâncias de parâmetros na fórmula. Essa seção lida com as duas principais questões relacionadas com essa premissa:

- como a abordagem proposta gera uma fatoração da fórmula?
- por que a fatoração pode levar a fórmulas simples?

A fatoração é alcançada por meio de uma parametrização hierárquica do modelo. Em um nível mais alto, atividades são independentes, então a modelagem paramétrica pode ser realizada. Os diagramas de sequência associados com cada atividade utilizam as mesmas premissas da abordagem apresentada nos Capítulos 3 e 4, assim, eles podem ser modelados da forma apresentada anteriormente. A abordagem composicional propõe uma maneia de dividir o modelo da LPS em modelos menores representando diagramas de sequência agrupados por atividades para então recombiná-los. Esse agrupamento representa uma fatoração parcial obtida na abordagem composicional.

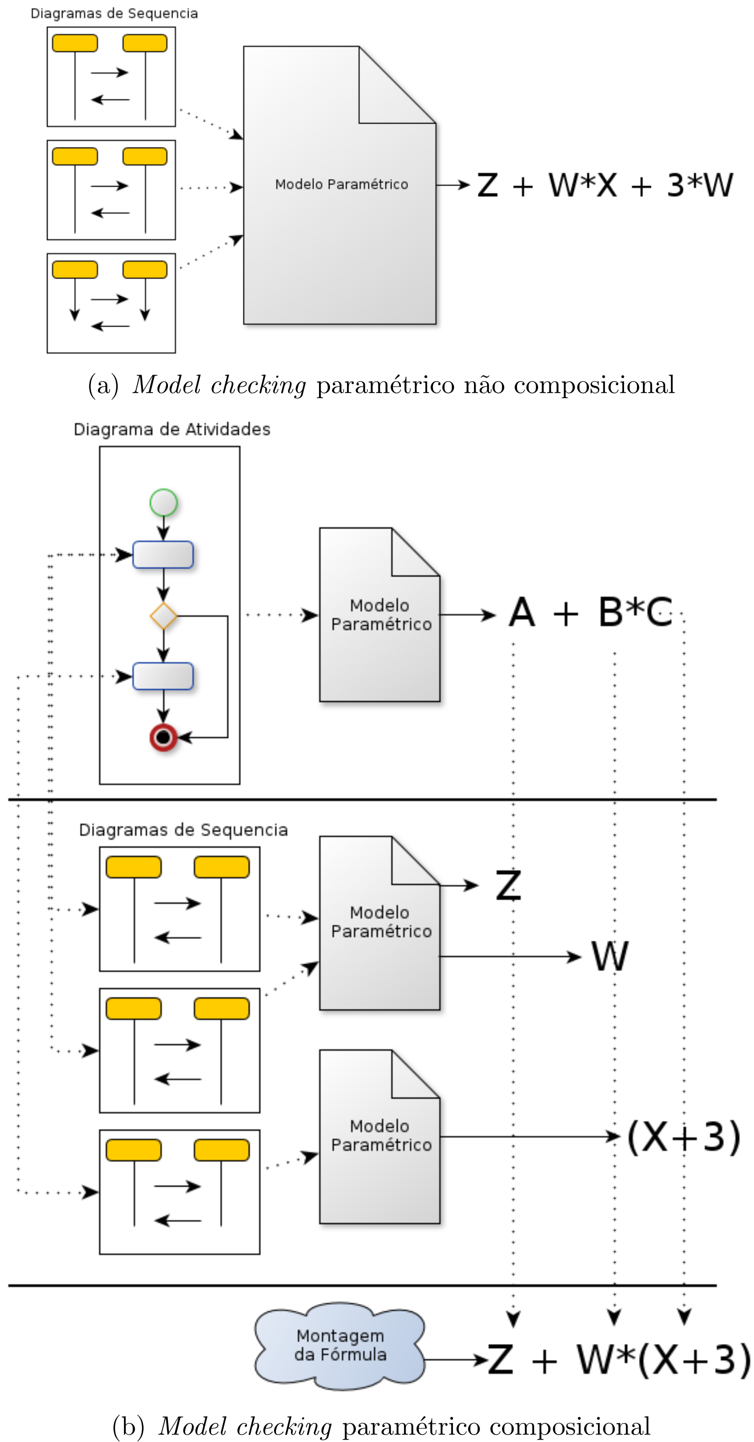
A Fig. 5.4(a) e 5.4(b) apresenta exemplos da abordagem não composicional apresentada nos Capítulos 3 e 4 e abordagem composicional para a mesma LPS. Z, W, X são parâmetros associados com *features* não obrigatórias de uma LPS.

Notavelmente a abordagem não composicional possui um processo simplificado de obtenção da fórmula paramétrica com menos passos e menos modelos. Uma das motivações dessa abordagem é construir um único modelo para representar toda a LPS, apesar da abordagem composicional requerer mais modelos esse número não está atrelado ao número de configurações da LPS.

Na abordagem composicional, a fórmula obtida a partir do diagrama de atividades $(A + BC)$ representa uma fatoração da fórmula apresentada na Fig. 5.4(a) $(Z + W * X + 3 * W)$ e as fórmulas obtidas por meio dos diagramas de sequência são mapeadas para parâmetros da fórmula fatorada.

Esse exemplo ilustra como a fatoração pode reduzir o tamanho da fórmula. A fórmula apresenta na Fig. 5.4(a) possui quatro operações (duas multiplicações e duas somas), a fórmula apresentada na Fig. 5.4(b) possui apenas três operações (duas somas e uma multiplicação). Adicionalmente, a segunda fórmula evita a repetição de parâmetros W o que reduz o esforço de avaliação desse parâmetro uma vez que este ocorre apenas uma vez. Note que o número de operações, assim como o número de operadores são expressos explicitamente nas fórmulas e o número real de operações executadas depende a estratégia de avaliação.

Observe que as fórmulas geradas por ambas as abordagens utilizam os mesmos parâmetros e resultam nos mesmos valores para uma mesma avaliação de parâmetros.



(a) *Model checking* paramétrico não composicional

(b) *Model checking* paramétrico composicional

Figura 5.4: Comparação das abordagens

5.3 Avaliação

Para avaliar a abordagem composicional esta foi comparada com a abordagem não composicional anteriormente proposta analisando aspectos quantitativos e qualitativos de ambas. A abordagem composicional proposta é vista como uma extensão para a não composicio-

nal uma vez que aquela é utilizada para construir modelos para esta. De agora em diante a abordagem de *model checking* paramétrico composicional será denotada por **CPMC** e a não composicional por **NCPMC**.

Para coletar os dados necessários para analise foi realizado um estudo de caso com as abordagens CPMC e NCPMC em LPSs acadêmicas.

5.3.1 Análise Quantitativa

A análise quantitativa consiste de um estudo de caso conduzido utilizado o método GQM (*Goal, Question, Metric*)(Basili et al., 1994). Esse método refina os objetivos de pesquisa em questões que por sua vez são refinadas para as métricas adequadas. As métricas respondem às questões que por sua vez ajudam a alcançar o objetivo de pesquisa.

Objetivo

A Tabela 5.1 apresenta o objetivo de pesquisa dessa análise quantitativa.

Objetivo	
Propósito	Comparar
Assunto	Tamanho
Objeto	Fórmula Paramétrica
Ponto de Vista	Engenheiro de Aplicação
Contexto	LPS

Tabela 5.1: Definição do Objetivo

Em suma, o objetivo do estudo de caso é comparar diferentes aspectos do tamanho da fórmula paramétrica gerada no contexto de uma LPS sob o ponto de vista do engenheiro de aplicação. Como dito anteriormente, o tamanho da fórmula é um problema quanto esta é avaliada com recursos limitados, assim, essa questão é uma responsabilidade do engenheiro de aplicação. Apesar da modelagem da LPS ser realizada pelo engenheiro de domínio, a avaliação da fórmula não é uma responsabilidade desse profissional.

Questões

O objetivo é refinado em duas questões para avaliar diferentes aspectos do tamanho da fórmula sob o ponto de vista do engenheiro de aplicação.

- Questão **Q1**: Há redução no tamanho textual da fórmula?
  - Essa questão avalia o tamanho da fórmula em seu formato texto uma vez que este representa o custo de realização do *parsing* ou de codificá-la diretamente no código fonte como discutido na Seção 3.5.2
- Questão **Q2**: Há redução no tamanho da fórmula?
  - Essa questão avalia o tamanho da fórmula com respeito a aspectos aritméticos.

Métricas

Antes de coletar os dados, cada métrica foi precisamente definida para evitar erros durante a coleta. Cada métrica da Tabela 5.2 é definida abaixo:

- Métrica**M1.1**: Número no texto da fórmula.
  - Número de bytes no arquivo texto da fórmula em formato ASCII. O texto ta fórmula é livre de caracteres de espaços, tabulações, quebras de linhas e de retorno de carro (*carriage return*).
- Métrica**M2.1**: Número de operandos.
  - Número de constantes mais o número de parâmetros na fórmula.
- Métrica**M2.2**: Número de operadores.
  - Número de ocorrências de caracteres $+, *, \hat{}, -, /$, respectivamente: soma, multiplicação, potenciação, subtração e divisão.
- Métrica**M2.3**: Número de parâmetros.
  - Qualquer sequência de caracteres alfabéticos delimitados por dois dois seguintes caracteres $+, *, \hat{}, -, /,), ($, ou seja, qualquer ocorrência de uma parâmetro, inclusive repetições.
- Métrica**M2.4**: Número de constantes.
  - Qualquer sequência numérica de caracteres opcionalmente com um '.' (ponto) na sequência delimitado por dois dos seguintes caracteres $+, *, \hat{}, -, /,), ($.

A Tabela 5.2 apresenta o mapeamento entre o objetivo, questões e métricas.

Questions	**Metrics**
Q1	M1.1
Q2	M2.1
	M2.2
	M2.3
	M2.4

Tabela 5.2: Questões e Métricas

Preparação do Estudo de Caso

O estudo de caso consiste em modelar a LPS com ambas as abordagens. As LPSs modeladas foram as seguintes:

1. Vital Signal Monitoring System: LPS dinâmica no domínio de monitoramento de sinais vitais. Essa LPS possui 16 *features*, 9 delas são *features* `OR`, 2 são *features* `Alternativas` e 5 são obrigatórias. Essa LPS dinâmica representa uma implementação real de outro trabalho contexto do nosso grupo de trabalho.
2. Mobile Media: LPS de uma aplicação para Java Mobile para gerenciar música, vídeo e fotos (Figueiredo et al., 2008). Essa LPS possui 12 *features*, 3 delas são *features* `Opcionais` e 9 obrigatórias.

Cada uma dessas LPS foi modelada utilizado as abordagnes NCPMC e CPMC. Os modelos de *features*, diagramas de sequência, diagramas de atividades, expressões PCTL e modelos paramétricos para a LPS *Vital Signal Monitoring System* estão disponíveis no apêndiceA. Os recursos relacionados com a LPS Mobile Media juntamente com as fórmulas da LPS *Vital Signal Monitoring System* estão disponíveis online [1].

Dado que apenas *features* não obrigatórias possuem parâmetros associados no modelo o *Vital Signal Monitoring System* possui 11 parâmetros distintos e o Mobile Media possui 3.

Antes da composição da fórmula (passo 5 da Fig 5.2), os modelos no nível de componente que resultaram em constantes foram operados para um único termo com 20 dígitos significativos de precisão.

A simulação foi conduzida com a ferramenta PARAM versão 2.2 64-bit em um sistema operacional Linux 64-bit Linux (Hahn et al., 2010). Todas as métricas foram coletadas automaticamente por meio de software e/ou scripts bash.

Análise dos Dados do Estudo de Caso

A Fig. 5.5 apresenta dos dados coletados durante o estudo de caso em gráficos. Os primeiros dois gráficos apresentam a métrica para primeira questão da Tabela 5.2, os dois últimos gráficos apresentam as métricas da segunda questão da Tabela 5.2. A Tabela 5.3 apresenta os dados coletados para cada métrica, LPS e abordagem de modelagem.

	Mobile Media		Vital Signal Monitoring System	
	NCPMC	CPMC	NCPMC	CPMC
M1.1	1439	992	1613626	162575
M2.1	21	49	60031	5819
M2.2	21	53	56575	5804
M2.3	12	9	43824	4064
M2.4	9	40	16207	1755

Tabela 5.3: Dados Coletados

Analisando os dados coletados relacionados com a métrica da primeira questão, verificou-se uma redução no tamanho textual para ambas as LPS. Entretanto, a redução da fórmula para LPS Mobile Media é devida a diferenças de precisão na representação das constantes e não representa uma redução real (como dito anteriormente, na preparação do estudo de

[1]https://code.google.com/p/spl-parametric-model-checking

caso as fórmulas compostas apenas por constantes foram avaliadas para um número real com 20 casas de precisão). Isso pode ser verificado diretamente nas fórmulas geradas e é reforçado por meio dos dados das demais métricas (número de operadores e operandos). Por outro lado, a LPS Vital Signal Monitoring System apresentou uma redução substancial no tamanho da fórmula e, assim com a LPS Mobile Media, esse resultado é reforçado pelas demais métricas coletadas.

As métricas de tamanho da fórmula demonstrar que o tamanho da fórmula para a LPS Vital Signal Monitoring System foi bastante reduzido (quase 90% de redução) em todas as métricas ao se utilizar a abordagem composicional , por outro lado a fórmula para a LPS Mobile Media aumentou em praticamente todas as métricas. A única métrica que apresentou redução para LPS Mobile Media foi o número de parâmetros, o que era esperado uma vez que a fórmula fatorada leva a uma menor redundância de parâmetros.

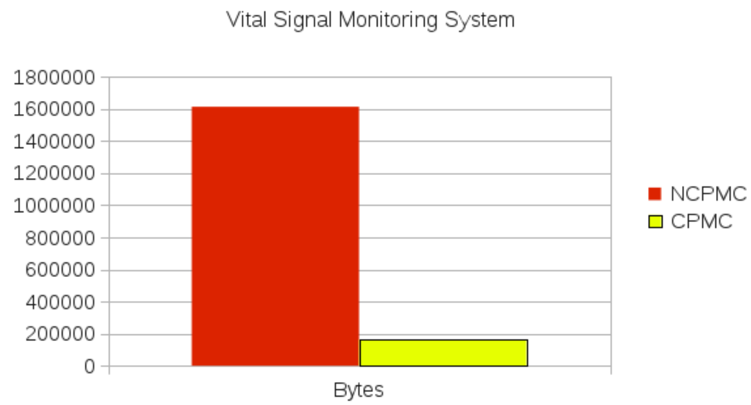
Apesar da LPS Vital Signal Monitoring System ter apresentado redução com o uso da abordagem CPMC, os resultados indicam que a abordagem não reduz a fórmula em todos os casos. A abordagem CPMC também reduziu a redundância na ocorrência de parâmetros na fórmula da LPS Mobile Media, mas mesmo com essa redução a fórmula gerada acabou sendo maior. Isso ocorre por que, como efeito colateral, a fatoração aumenta a redundância das constantes na fórmula da LPS Mobile Media.

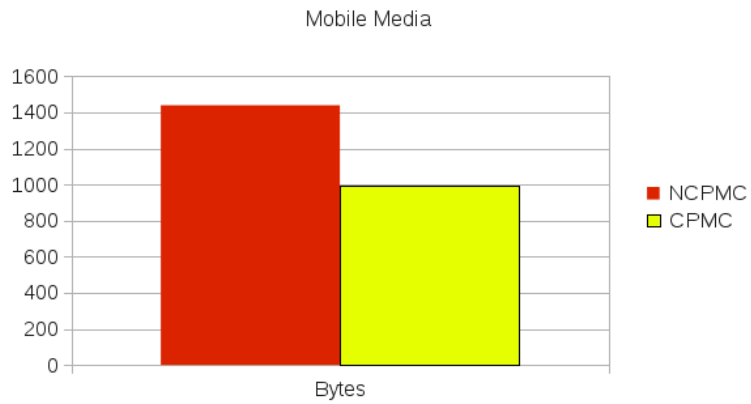
Observe que o objetivo da abordagem era reduzir o tamanho da fórmula por meio da redução da redundância de parâmetros e que para ambos os casos a quantidade de ocorrências de parâmetros foi reduzida. Entretanto, foi observado que a redundância na ocorrência dos parâmetros não é suficiente para diminuir a fórmula em todos os casos como foi exemplificado pela LPS Mobile Media

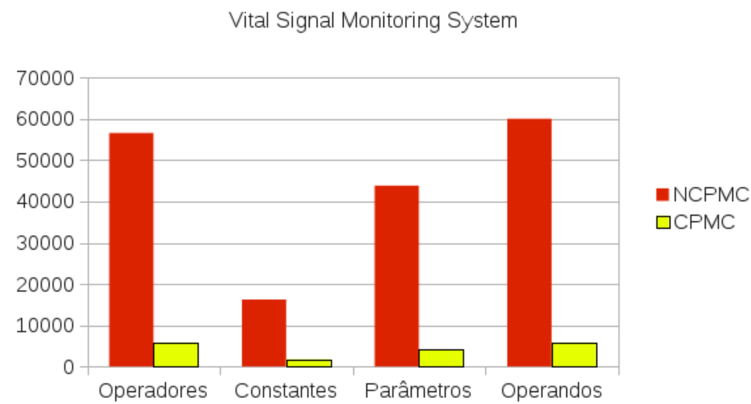
Mobile Media é uma LPS pequena com poucas *features* variáveis (apenas 3) e, por isso, a redundância de parâmetros não afeta o tamanho fórmula como a redundância de constantes. No formato completamente expandido, todas as contantes são agregadas em uma única constante possivelmente reduzindo o tamanho da fórmula (como discutido na Seção 3.5.1).

O número mínimo de ocorrências para cada parâmetro é um. Assim, o número mínimo de parâmetros na fórmula da LPS Vital Signal Monitoring System é 11 e para a Mobile Media é 3. Entretanto, o número máximo de ocorrências é ilimitado uma vez que a fórmula completamente expandida pode ter qualquer termo composto pela multiplicação de qualquer combinação de parâmetros e constantes. Dado que a fórmula é um polinômio, esta pode ter um parâmetro com um expoente arbitrário, logo não há um número máximo de ocorrências para o parâmetro na fórmula (possivelmente infinitas ocorrências). Como resultado, modelos grandes com muitos parâmetros tendem a apresentar altar redução no tamanho da fórmula por meio de fatoração.

Apesar dos resultados não serem conclusivos, eles indicam que a redução da fórmula é mais perceptível em LPS grandes com mais *features* variáveis. Entretanto, a abordagem CPMC é aplicável a qualquer LPS uma vez que o resultado fatorado pode ser expandido para o formato completamento expandido aplicando a propriedade distributiva o que dá flexibilidade para o engenheiro de aplicação da LPS.







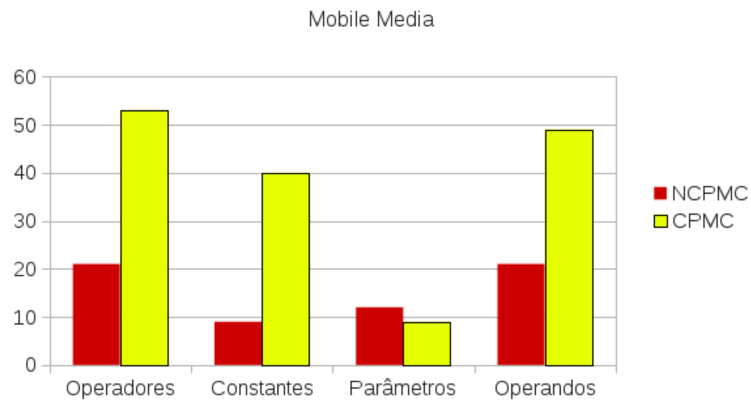


Figura 5.5: Gráficos de Simulação

5.3.2 Limitações e Ameaças à Validade

A abordagem composicional proposta não é capaz de lidar com variabilidade no modelo no nível de atividades, e não é possível garantir que todos os sistemas podem ser representados de tal forma que o diagrama seja adequado para a abordagem proposta e livre de variabilidade (isto é, que não haja necessidade de representar a variabilidade no nível de atividades). Ademais, a documentação necessária pode não estar disponível ou difícil de obter, o que reduz a aplicabilidade da abordagem.

Adicionalmente, a abordagem proposta é especificamente projetada para cálculo de confiabilidade. Apesar de poder avaliar outras propriedades não funcionais , esse aspecto não foi explorado. Além disso, as expressões PCTL utilizadas limitam-se ao uso da propriedade de *reachability* em tempo ilimitado para determinar os estados de sucesso. É provável que expressões outras expressões sejam possíveis, porém este trabalho não as avaliou.

O estudo de caso foi conduzido em um conjunto pequeno de LPS com um número de *features* relativamente pequeno, um estudo de caso mais abrangentes com mais LPS e mais variabilidade pode levar a resultados mais precisos.

Em particular, as ameaças à validade foram tratadas da seguinte forma:

- A validade de construção diz respeito ao correto estabelecimento de métricas operacionais para os conceitos estudados. Diversas métricas foram selecionadas para cobrindo diferentes aspectos relacionados ao tamanho da fórmula. Essas métricas foram refinadas no contexto de experiência prática de avaliação de fórmulas grandes em tempo de execução.

- A validade interna estabelece uma relação causal onde é mostrado que determinadas condições levam a outras condições. A abordagem composicional não resulta em fórmulas pequenas para qualquer caso. A LPS Mobile Media tem menos parâmetros em seu modelo que a LPS Vital Signal Monitoring. Apesar das diferentes na redução da fórmula estarem associadas com o número de parâmetros, outros aspectos podem impactar o resultado e a investigação de tais aspectos está fora do escopo desse trabalho.

- A validade externa diz respeito ao domínio no qual os resultados podem ser generalizados. Foram comparadas as abordagens composicionais e não composicionais no contexto de duas LPS pequenas, porém representativas, que exemplificaram situações onde a fatoração pode levar a redução ou aumento do tamanho da fórmula.

- A confiabilidade diz respeito à reprodutibilidade do estudo com os mesmos resultados. A documentação utilizada para o Mobile Media foi obtida por meio de engenharia reversa. Esta consistiu em grande parte de atividades manuais. Para mitigar às ameaças relacionadas à reprodutibilidade todos os artefatos utilizados no estudo de caso, inclusive a documentação obtida por meio de engenharia reversa, estão disponíveis online em `https://code.google.com/p/spl-parametric-model-checking/`.

5.4 Trabalhos Relacionados

Até onde é sabido, o trabalho mais relacionado à essa proposta lida com model checking incremental de LPSs (Cordy et al., 2012). Essa técnica permite que sejam feitas mudanças nas *features* de uma LPS sem a necessidade de refazer o *model checking* de todo o modelo. Por meio de rastreamento de dependências de *features*, a técnica é capaz de terminar que parte do modelo precisa ser recalculada. Essa técnica só pode ser aplicada quando da adição de *features* conservativas, isto é, aquelas que não removem comportamento do sistema. Outros tipos de mudanças não são tratados.

Na abordagem composicional proposta, é também possível obter as vantagens de um *model checking* incremental em determinados casos. Apesar de não estabelecer restrições em relação aos tipos de mudanças nas *features* (conservativas e não conservativas), as mudanças na variabilidade precisam ser rastreadas desde o modelo de *features* até a documentação de maneira a determinar que diagrama de sequência foi afetado e determinar quais modelos precisam ser recalculados.

5.5 Conclusão

Em alguns cenários, o tamanho da fórmula é uma questão importante, em particular cenários onde é necessário avaliar a fórmula em tempo de execução com recursos computacionais limitados. Esse trabalho apresenta uma abordagem paramétrica composicional que é capaz de reduzir o tamanho da fórmula por meio da obtenção de uma fatoração parcial da fórmula. Essa fatoração pode reduzir o tamanho da fórmula por meio da redução da redundância na ocorrência dos parâmetros.

A abordagem proposta pode levar a reduções de quase 90% em número de operandos. Em alguns casos a abordagem pode levar a fórmulas grandes, entretanto a fórmula completamente expandida (formato obtido pela abordagem não composicional) pode ser obtida por meio da aplicação da propriedade distributiva na fórmula fatorada. Assim, a abordagem proposta pode ser utilizada em qualquer caso e caberia ao engenheiro de domínio decidir se a fatoração parcial obtida é adequada ao uso desejado ou se é preciso expandir a fórmula até seu formato completamente expandido.

Capítulo 6

Conclusão

Construir software confiável é uma tarefa importante, especialmente para aqueles de natureza crítica, ou seja, aqueles cujas falhas podem levar a consequências desastrosas. Estimar a confiabilidade de um software não é uma tarefa fácil, em particular o uso de model checking requer que o modelo apresente um nível adequado de detalhamento que torne viável a utilização de tal técnica.

Esse problema é ainda mais crítico quando lidamos não apenas com um software, mas com uma família de software em uma LPS. Nesse caso, o esforço de construção dos modelos dos diversos produtos cresce exponencialmente com o número de características da LPS. Assim, uma abordagem escalável em relação à quantidade de modelos produzidos é uma necessidade. Nesse contexto, algumas abordagens endereçam diretamente esse problema entretanto as abordagens existentes apresentam restrições em termos do tipo de variabilidade e da representação destas na arquitetura.

Neste trabalho apresentamos uma estratégia de modelagem de LPS capaz de representar qualquer tipo de variabilidade. Por meio do uso de *model checking* paramétrico é possível construir um único modelo capaz de representar todos os produtos da LPS. Assim, por meio de diferentes valorações dos parâmetros do modelo é possível obter o valor estimado de confiabilidade dos diversos produtos.

Para desenvolver essa técnica apresentamos os principais conceitos de *model checking* paramétrico e como esses se relacionam com a modelagem de variabilidade. A partir desse estudo propomos uma técnica capaz de representar qualquer tipo de variabilidade por meio do tratamento de características opcionais. Em seguida apresentamos uma prova da correção da técnica demonstrando como esta é capaz de seletivamente isolar caminhos da cadeia de Markov sintetizada a partir da modelagem paramétrica.

A fórmula parametrizada permite diferentes utilizações para o cálculo de confiabilidade. Em determinados cenários o tamanho da fórmula obtida por meio do cálculo paramétrico é relevante, em particular, cenários onde a disponibilidade de recursos computacionais é limitada. Nesses cenários o tamanho da fórmula deve ser tal que o custo de avaliação seja viável. Assim, mitigar o tamanho da fórmula final obtida pela técnica é uma necessidade.

Neste trabalho, discutimos fatores que levam ao crescimento da forma que servem como guia para a modelagem de forma a reduzir o tamanho da fórmula gerada. Apresentamos ainda, um estudo da composicionalidade de modelos paramétricos como forma de se reduzir o tamanho da fórmula e consequentemente o esforço de avaliação da mesma.

6.1 Trabalhos Futuros

O método proposto deixa em aberto alguns pontos que podem ser explorados em trabalhos futuros, são eles:

- Desenvolver uma notação de forma que a variabilidade possa ser expressa diretamente nos diagramas UML de maneira anotativa.
- Automação do método de modelagem utilizando diagramas anotados com variabilidade e o modelo de características como entradas. A construção automatizada pode dispor de técnicas de otimização do modelo de forma reduzir o tamanho da fórmula gerada.
- Demonstrar a correção do funcionamento da abordagem composicional.
- Estender a abordagem composicional de forma que esta seja capaz de tratar variabilidade no nível de diagrama de atividades.
- Expandir o estudo de caso da abordagem composicional para incluir novas métricas possivelmente considerando o custo de avaliação das diferentes operações aritméticas para uma determinada arquitetura de processador.
- Modelar formalmente em ambientes de prova as demonstrações apresentadas.

Apêndice A

Simulação Vital Signal Monitoring System

Neste apêndice apresentamos a documentação utilizada na construção dos modelos composicionais e não composicionais discutidos no Capítulo 5 bem como os modelos e expressões PCTL utilizados. Ao final apresentamos as fórmulas obtidas para a abordagem composicional, porém omitimos a fórmula da abordagem não composicional devido a seu tamanho.

Os diagramas UML de sequência apresentados neste apêndice apresentam fragmentos de interação do tipo `Opt` como forma de representar partes variáveis dos diagramas. Essas partes correspondem a pontos de variabilidades das características do modelo de características. Note que essa notação não expressa nos diagramas o tipo de restrição entre as características (Opcionais, Alternativas, OR, Obrigatórias), apenas denotam que a parte delimitada pelo fragmento pode fazer ou não parte do diagrama de acordo com a seleção das características. As restrições com relação à seleção das características é expressa no modelo de características. A notação descrita não foi estudada e não sabemos, a princípio, sua aplicabilidade em outros contextos, entretanto a mesma se mostrou adequada para a documentação utilizada nessa simulação.

A.1 Documentação

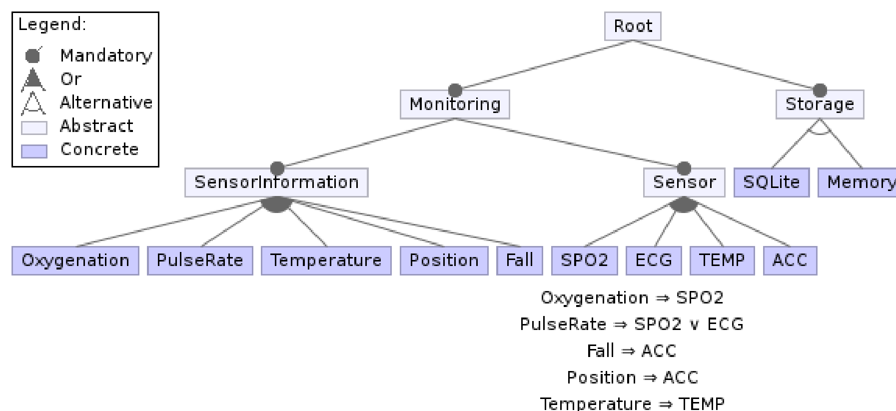


Figura A.1: Modelo de Características

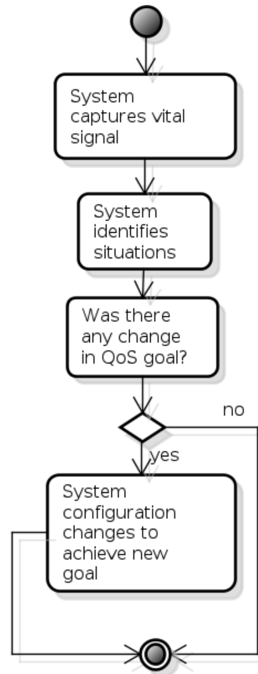


Figura A.2: Diagrama de Atividades

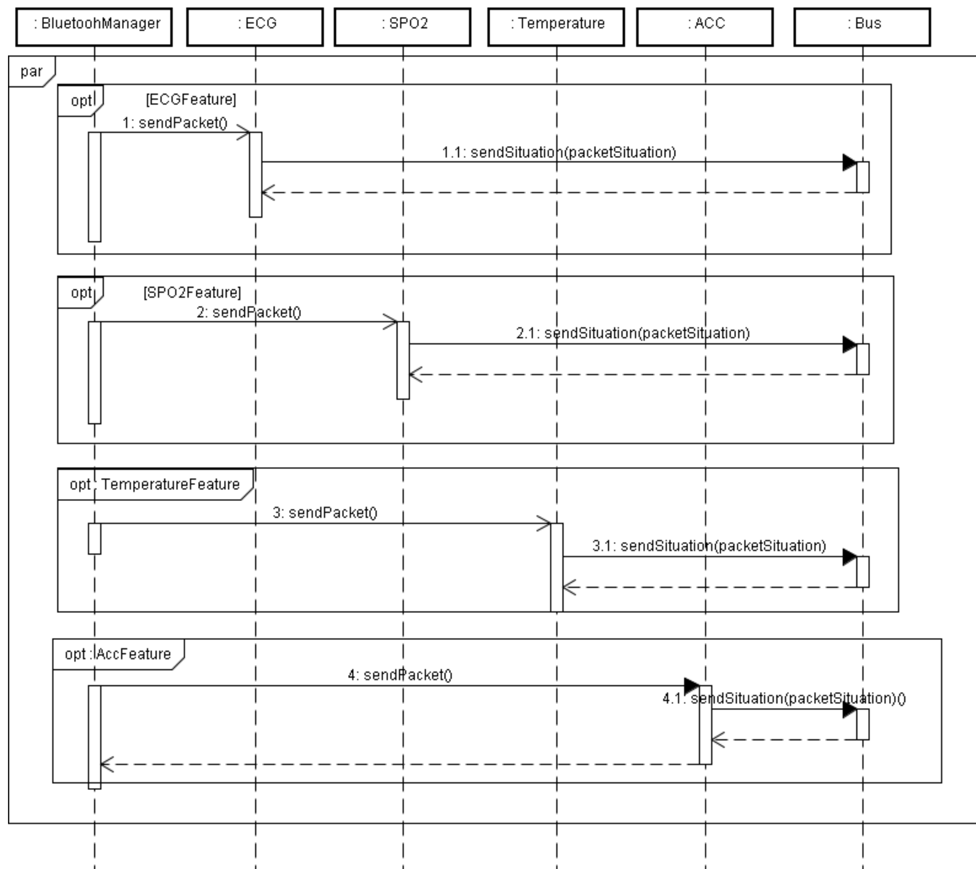


Figura A.3: Diagrama de sequência (Atividade: *System captures vital signal*)

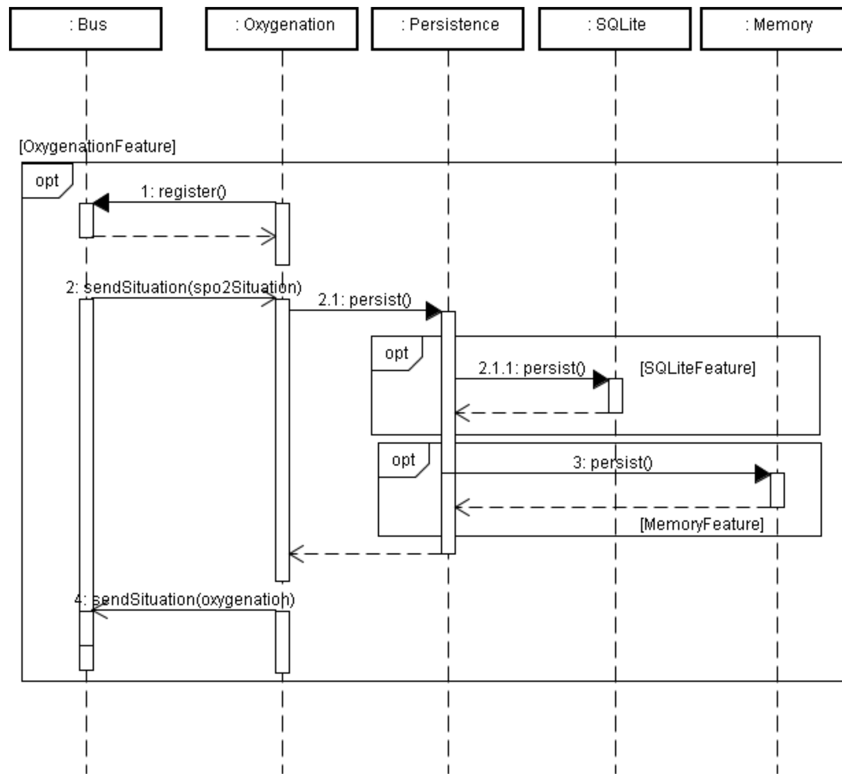


Figura A.4: Diagrama de sequência (Atividade: *System identifies situations*)

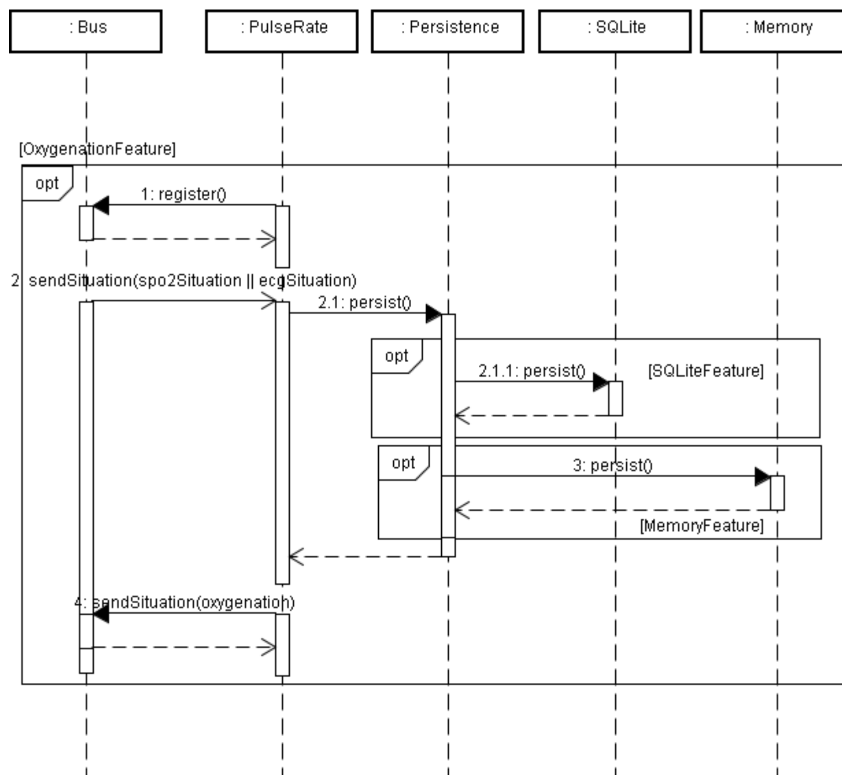


Figura A.5: Diagrama de sequência (Atividade: *System identifies situations*)

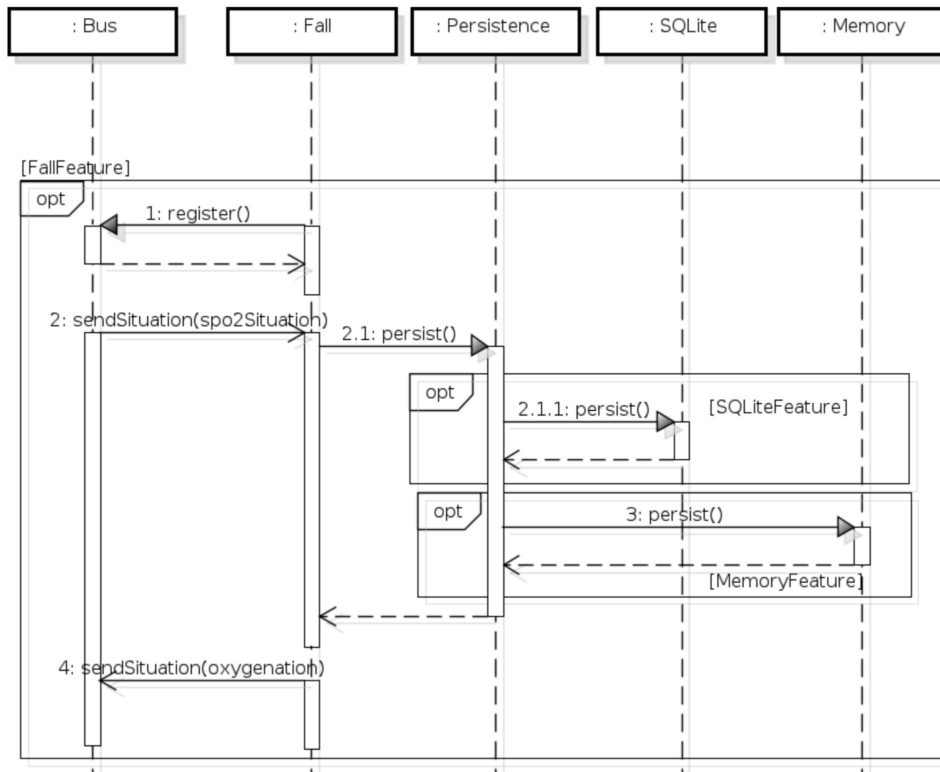


Figura A.6: Diagrama de sequência (Atividade: *System identifies situations*)

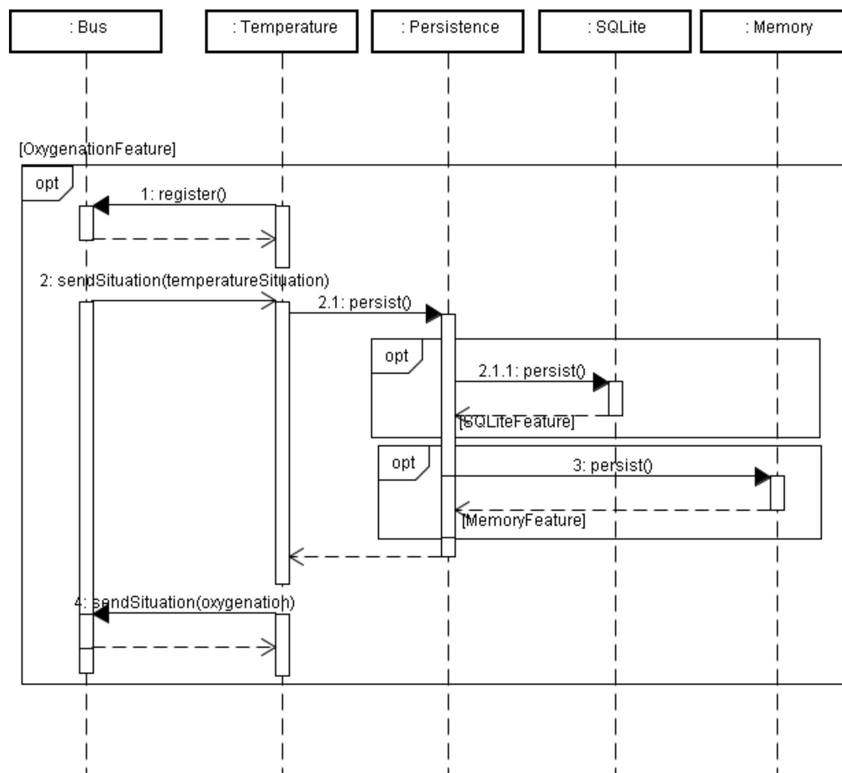


Figura A.7: Diagrama de sequência (Atividade: *System identifies situations*)

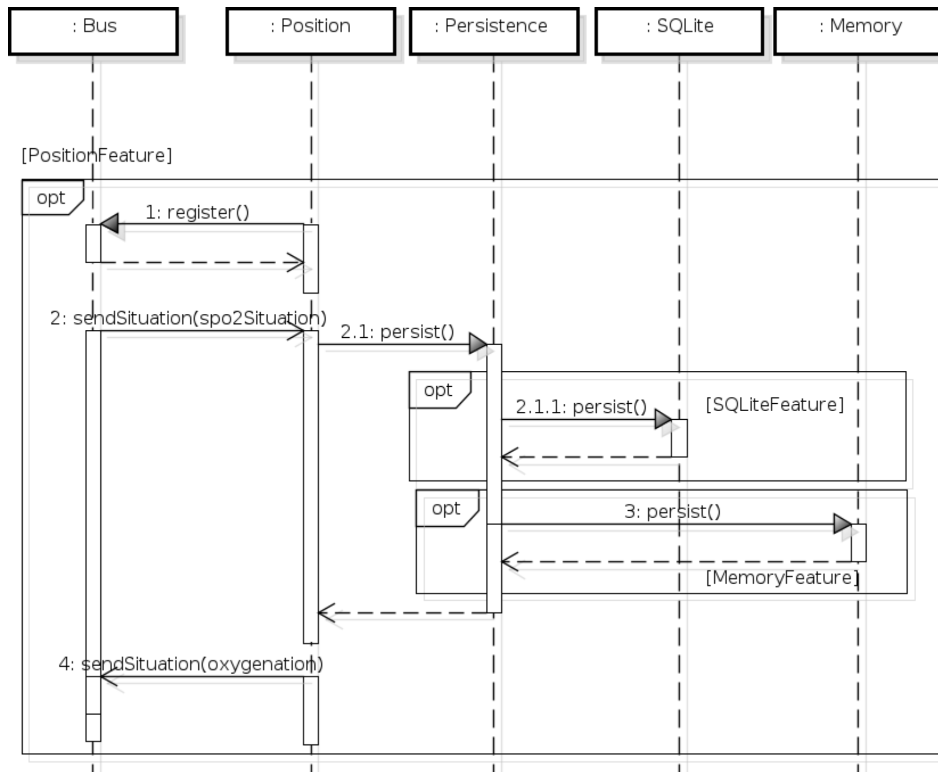


Figura A.8: Diagrama de sequência (Atividade: *System identifies situations*)

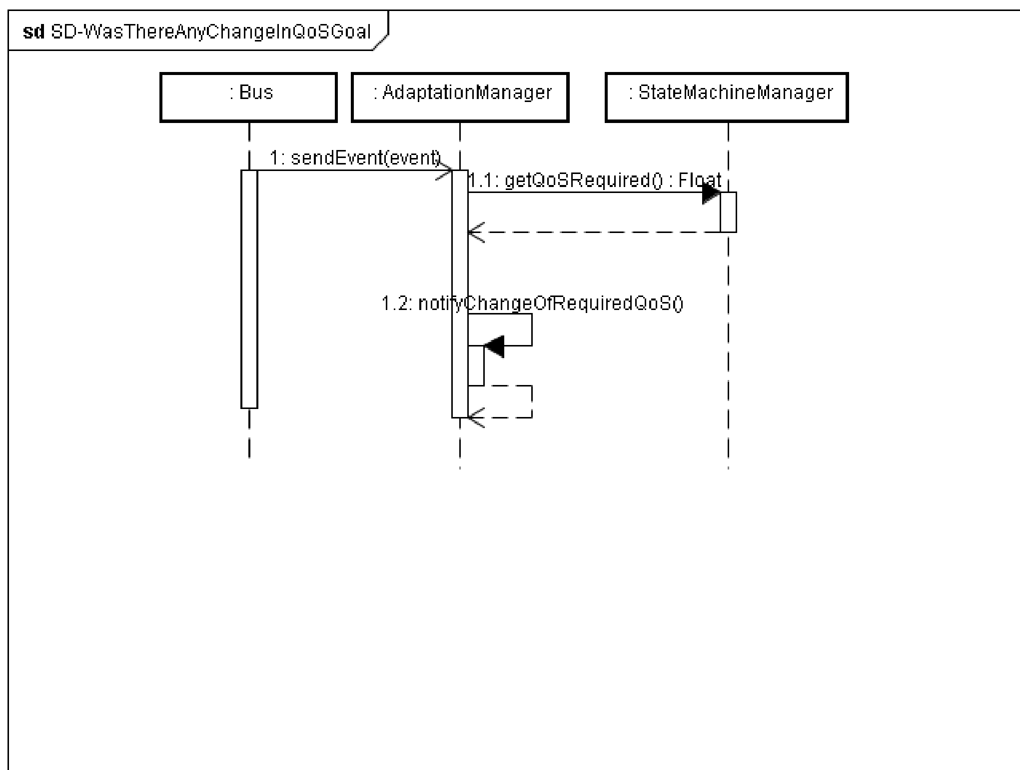


Figura A.9: Diagrama de sequência (Atividade: *Was there any change in QoS goal?*)

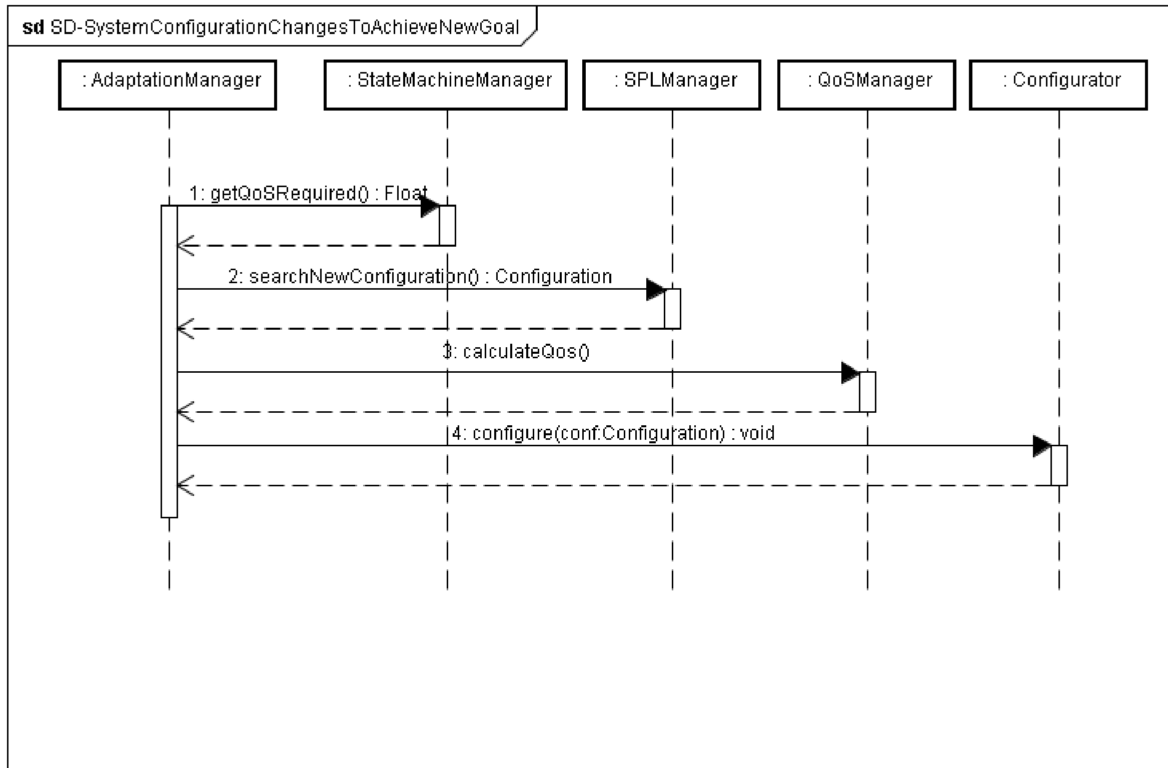


Figura A.10: Diagrama de sequência (Atividade: *System configuration changes to achieve new goal*)

A.2 Modelos

A.2.1 Não composicional

Listagem A.1: Modelo PARAM

```
dtmc
param int fSSPO2;
param int fSTemp;
param int fSECG;
param int fSACC;
param int fOxy;
param int fTemp;
param int fPlsRt;
param int fPos;
param int fFall;
param int fMem;
param int fSqlite;
//Reliability
const double r = 0.999;
const double rBM = r;
const double rSECG = r;
const double rSSPO2 = r;
const double rSTemp = r;
const double rSAcc = r;
const double rBus = r;
const double rOxy = r;
const double rTemp = r;
const double rPlsRt = r;
const double rPos = r;
const double rFall = r;
const double rPersistence = r;
const double rSqlite = r;
const double rMem = r;
const double rAM = r;
const double rStateMachine = r;
const double rSPLManager = r;
const double rQosManager = r;
const double rConfigurator = r;
//Usage Profile
```

```
const double reconfigProfile = 0.5;
formula NE = (s0!=0 & se!=0 & ss!=0 & st!=0 & sa!=0 & sb!=0 & sp!=0 & so!=0 & su!=0 & sn!=0
 & sf!=0 & sr!=0 & sq!=0 & sm!=0 & si!=0 & sg!=0 & sx!=0 & sy!=0 & sz!=0);
module BluetoothManager
        s0 : [0..3] init 1;
        pECG : [0..1] init 0;
        pSPO2 : [0..1] init 0;
        pTEMP: [0..1] init 0;
        pACC: [0..1] init 0;
        [fails0] s0 = 0 -> (s0'=0);//Fail State
        [] s0 = 1 -> (1-fSSPO2)*(fSECG) *(1-fSTemp)*(1-fSACC) : (s0'=2)&(pECG'=1)
               + (fSSPO2) *(1-fSECG)*(1-fSTemp)*(1-fSACC) : (s0'=2)&(pSPO2'=1)
               + (1-fSSPO2)*(1-fSECG)*(fSTemp) *(1-fSACC) : (s0'=2)&(pTEMP'=1)
               + (fSSPO2) *(fSECG) *(1-fSTemp)*(1-fSACC) : (s0'=2)&(pECG'=1)&(pSPO2'=1)
               + (1-fSSPO2)*(fSECG) *(fSTemp) *(1-fSACC) : (s0'=2)&(pECG'=1)&(pTEMP'=1)
               + (fSSPO2) *(1-fSECG)*(fSTemp) *(1-fSACC) : (s0'=2)&(pSPO2'=1)&(pTEMP'=1)
               + (fSSPO2) *(fSECG) *(fSTemp) *(1-fSACC) : (s0'=2)&(pSPO2'=1)&(pTEMP'=1)&(pECG'=1)
               + (1-fSSPO2)*(1-fSECG)*(1-fSTemp)*(1-fSACC) : (s0'=0)
               + (1-fSSPO2)*(fSECG) *(1-fSTemp)*(fSACC) : (s0'=2)&(pECG'=1)&(pACC'=1)
               + (fSSPO2) *(1-fSECG)*(1-fSTemp)*(fSACC) : (s0'=2)&(pSPO2'=1)&(pACC'=1)
               + (1-fSSPO2)*(1-fSECG)*(fSTemp) *(fSACC) : (s0'=2)&(pTEMP'=1)&(pACC'=1)
               + (fSSPO2) *(fSECG) *(1-fSTemp)*(fSACC) : (s0'=2)&(pECG'=1)&(pSPO2'=1)&(pACC'=1)
               + (1-fSSPO2)*(fSECG) *(fSTemp) *(fSACC) : (s0'=2)&(pECG'=1)&(pTEMP'=1)&(pACC'=1)
               + (fSSPO2) *(1-fSECG)*(fSTemp) *(fSACC) : (s0'=2)&(pSPO2'=1)&(pTEMP'=1)&(pACC'=1)
               + (fSSPO2) *(fSECG) *(fSTemp) *(fSACC) : (s0'=2)&(pSPO2'=1)&(pTEMP'=1)&(pECG'=1)&(pACC'=1)
               + (1-fSSPO2)*(1-fSECG)*(1-fSTemp)*(fSACC) : (s0'=2)&(pACC'=1);
        [sendPacket] s0 = 2 & pECG=1 & NE -> (rSECG) : (s0'=3) + (1 - rSECG) : (s0'=0);
        [sendPacket] s0 = 2 & pSPO2=1 & NE-> (rSSPO2) : (s0'=3) + (1 - rSSPO2) : (s0'=0);
        [sendPacket] s0 = 2 & pTEMP=1 & NE-> (rSTemp) : (s0'=3) + (1 - rSTemp) : (s0'=0);
        [sendPacket] s0 = 2 & pACC =1 & NE-> (rSAcc) : (s0'=3) + (1 - rSAcc) : (s0'=0);
endmodule
module ECG
        se : [0..3] init 1;
        se1 : [0..1] init 0;
        [failse] se = 0 -> (se'=0);//Fail State
        [sendPacket] se = 1 & NE-> (se'=2);
        [sendSituationSe] se = 2 & NE & (sa1 = 0 & st1 = 0 & ss1 = 0 & se1 = 0) -> (rBus) : (se'=2) & (se1'=1)
        + (1 - rBus) : (se'=0);
        [sendPacket_returnSe] se = 2 & NE ->(rSECG) : (se'=3) + (1 - rSECG) : (se'=0);
endmodule
module SPO2
        ss : [0..3] init 1;
        ss1 : [0..1] init 0;
        [failss] ss = 0 -> (ss'=0);//Fail State
        [sendPacket] ss = 1 & NE -> (ss'=2);
        [sendSituationSs] ss = 2 & NE & (sa1 = 0 & st1 = 0 & ss1 = 0 & se1 = 0) -> (rBus) : (ss'=2) & (ss1'=1) +
         (1 - rBus) : (ss'=0);
        [sendPacket_returnSs] ss = 2 & NE -> (rSSPO2) : (ss'=3) + (1 - rSSPO2) : (ss'=0);
endmodule
module Temp
        st : [0..3] init 1;
        st1 : [0..1] init 0;
        [failst] st = 0 -> (st'=0);//Fail State
        [sendPacket] st = 1 & NE -> (st'=2);
        [sendSituationSt] st = 2 & NE & (sa1 = 0 & st1 = 0 & ss1 = 0 & se1 = 0) -> (rBus) : (st'=2) & (st1'=1) +
         (1 - rBus) : (st'=0);
        [sendPacket_returnSt] st = 2 & NE ->(rSTemp) : (st'=3) + (1 - rSTemp) : (st'=0);
endmodule
module ACC
        sa : [0..3] init 1;
        sa1 : [0..1] init 0;
        [failsa] sa = 0 -> (sa'=0);//Fail saate
        [sendPacket] sa = 1 & NE -> (sa'=2);
        [sendSituationSa] sa = 2 & NE & (sa1 = 0 & st1 = 0 & ss1 = 0 & se1 = 0) -> (rBus) : (sa'=2) & (sa1'=1) +
         (1 - rBus) : (sa'=0);
        [sendPacket_returnSa] sa = 2 & NE -> (rSAcc) : (sa'=3) + (1 - rSAcc) : (sa'=0);
endmodule
module Bus
        sb : [0..37] init 1;
        [failsb] sb = 0 -> (sb'=0);//Fail saate
        [sendPacket_returnSa] sb = 1 & NE & (sa1 = 1 | st1 = 1 | ss1 = 1 | se1 = 1) -> (sb'=2);
        [sendPacket_returnSt] sb = 1 & NE & (sa1 = 1 | st1 = 1 | ss1 = 1 | se1 = 1) -> (sb'=2);
        [sendPacket_returnSs] sb = 1 & NE & (sa1 = 1 | st1 = 1 | ss1 = 1 | se1 = 1) -> (sb'=2);
        [sendPacket_returnSe] sb = 1 & NE & (sa1 = 1 | st1 = 1 | ss1 = 1 | se1 = 1) -> (sb'=2);
        [fPlsRt_Selection] sb = 2 & NE ->
                                (fPlsRt) : (sb'=3) +     //fPlsRt selected
                                (1 - fPlsRt) : (sb'=8);//fPlsRt deselected
        [registerPlsRt] sb = 3 & NE -> (sb'=4);
        [registerPlsRt_return] sb = 4 & NE -> (rPlsRt) : (sb'=5) + (1-rPlsRt) : (sb'=0);
        [sendEventPslRt] sb = 5 & NE -> (rPlsRt) : (sb'=6) + (1-rPlsRt) : (sb'=0);
        [sendSituationPslRt] sb = 6 & NE -> (sb'=7);
        [sendSituationPslRt_return] sb = 7 & NE -> (rPlsRt) : (sb'=8) + (1 - rPlsRt) : (sb'=0);
        [fOxy_Selection] sb = 8 & NE ->
                                (fOxy) : (sb'=9) +       //fOxy selected
                                (1 - fOxy) : (sb'=14);//fOxy deselected
        [registerOxy] sb = 9 & NE -> (sb'=10);
        [registerOxy_return] sb = 10 & NE -> (rOxy) : (sb'=11) + (1-rOxy) : (sb'=0);
        [sendEventOxy] sb = 11 & NE -> (rOxy) : (sb'=12) + (1-rOxy) : (sb'=0);
        [sendSituationOxy] sb = 12 & NE -> (sb'=13);
        [sendSituationOxy_return] sb = 13 & NE -> (rOxy) : (sb'=14) + (1 - rOxy) : (sb'=0);
        [fTemp_Selection] sb = 14 & NE ->
                                (fTemp) : (sb'=15) +     //fTemp selected
                                (1 - fTemp) : (sb'=20);//fTemp deselected
        [registerTemp] sb = 15 & NE -> (sb'=16);
        [registerTemp_return] sb = 16 & NE -> (rTemp) : (sb'=17) + (1-rTemp) : (sb'=0);
```

```
        [sendEventTemp] sb = 17 & NE -> (rTemp) : (sb'=18) + (1-rTemp) : (sb'=0);
        [sendSituationTemp] sb = 18 & NE -> (sb'=19);
        [sendSituationTemp_return] sb = 19 & NE -> (rTemp) : (sb'=20) + (1 - rTemp) : (sb'=0);
        [fPos_Selection] sb = 20 & NE ->
                              (fPos) : (sb'=21) +      //fPos selected
                              (1 - fPos) : (sb'=26);//fPos deselected
        [registerPos] sb = 21 & NE -> (sb'=22);
        [registerPos_return] sb = 22 & NE -> (rPos) : (sb'=23) + (1-rPos) : (sb'=0);
        [sendEventPos] sb = 23 & NE -> (rPos) : (sb'=24) + (1-rPos) : (sb'=0);
        [sendSituationPos] sb = 24 & NE -> (sb'=25);
        [sendSituationPos_return] sb = 25 & NE -> (rPos) : (sb'=26) + (1 - rPos) : (sb'=0);
        [fFall_Selection] sb = 26 & NE ->
                              (fFall) : (sb'=27) +     //fFall selected
                              (1 - fFall) : (sb'=32);//fFall deselected
        [registerFall] sb = 27 & NE -> (sb'=28);
        [registerFall_return] sb = 28 & NE -> (rFall) : (sb'=29) + (1-rFall) : (sb'=0);
        [sendEventFall] sb = 29 & NE -> (rFall) : (sb'=30) + (1-rFall) : (sb'=0);
        [sendSituationFall] sb = 30 & NE -> (sb'=31);
        [sendSituationFall_return] sb = 31 & NE -> (rFall) : (sb'=32) + (1 - rFall) : (sb'=0);
        [start_QosGoal] sb = 32 & NE -> (sb'=37);
        [sendEventAM] sb = 37 & NE -> (rAM) : (sb'=33) + (1 - rAM) : (sb'=0);
        [sendEventAM_return] sb = 33 & NE -> (sb'=34);
        [profile_Decision] sb = 34 & NE ->
                              (reconfigProfile) : (sb'=35) + //Reconfiguration Triggered
                              (1 - reconfigProfile) : (sb'=36);//End execution
        [start_Reconfiguration] sb = 35 & NE -> (sb'=35);
        [end_QosGoal] sb = 36 & NE -> (sb'=36);//END
endmodule
module PulseRate
        sp : [0..7] init 1;
        [failsp] sp = 0 -> (sp'=0);//Fail saate
        [registerPlsRt] sp = 1 & NE -> (rBus) : (sp'=2) + (1-rBus) : (sp'=0);
        [registerPlsRt_return] sp = 2 & NE -> (sp'=3);
        [sendEventPslRt] sp = 3 & NE -> (sp'=4);
        [persistPersistencePlsRt] sp = 4 & NE -> (rPersistence) : (sp'=5) + (1-rPersistence) : (sp'=0);
        [persistPersistencePlsRt_return] sp = 5 & NE -> (sp'=6);
        [sendSituationPslRt] sp = 6 & NE -> (rBus) : (sp'=7) + (1-rBus) : (sp'=0);
        [sendSituationPslRt_return] sp = 7 & NE -> (sp'=7);
endmodule
module Oxygenation
        so : [0..7] init 1;
        [failso] so = 0 -> (so'=0);//Fail saate
        [registerOxy] so = 1 & NE -> (rBus) : (so'=2) + (1-rBus) : (so'=0);
        [registerOxy_return] so = 2 & NE -> (so'=3);
        [sendEventOxy] so = 3 & NE -> (so'=4);
        [persistPersistenceOxy] so = 4 & NE -> (rPersistence) : (so'=5) + (1-rPersistence) : (so'=0);
        [persistPersistenceOxy_return] so = 5 & NE -> (so'=6);
        [sendSituationOxy] so = 6 & NE -> (rBus) : (so'=7) + (1-rBus) : (so'=0);
        [sendSituationOxy_return] so = 7 & NE -> (so'=7);
endmodule
module Temperature
        su : [0..7] init 1;
        [failsu] su = 0 -> (su'=0);//Fail suate
        [registerTemp] su = 1 & NE -> (rBus) : (su'=2) + (1-rBus) : (su'=0);
        [registerTemp_return] su = 2 & NE -> (su'=3);
        [sendEventTemp] su = 3 & NE -> (su'=4);
        [persistPersistenceTemp] su = 4 & NE -> (rPersistence) : (su'=5) + (1-rPersistence) : (su'=0);
        [persistPersistenceTemp_return] su = 5 & NE -> (su'=6);
        [sendSituationTemp] su = 6 & NE -> (rBus) : (su'=7) + (1-rBus) : (su'=0);
        [sendSituationTemp_return] su = 7 & NE -> (su'=7);
endmodule
module Position
        sn : [0..7] init 1;
        [failsn] sn = 0 -> (sn'=0);//Fail snate
        [registerPos] sn = 1 & NE -> (rBus) : (sn'=2) + (1-rBus) : (sn'=0);
        [registerPos_return] sn = 2 & NE -> (sn'=3);
        [sendEventPos] sn = 3 & NE -> (sn'=4);
        [persistPersistencePos] sn = 4 & NE -> (rPersistence) : (sn'=5) + (1-rPersistence) : (sn'=0);
        [persistPersistencePos_return] sn = 5 & NE -> (sn'=6);
        [sendSituationPos] sn = 6 & NE -> (rBus) : (sn'=7) + (1-rBus) : (sn'=0);
        [sendSituationPos_return] sn = 7 & NE -> (sn'=7);
endmodule
module Fall
        sf : [0..7] init 1;
        [failsf] sf = 0 -> (sf'=0);//Fail sfate
        [registerFall] sf = 1 & NE -> (rBus) : (sf'=2) + (1-rBus) : (sf'=0);
        [registerFall_return] sf = 2 & NE -> (sf'=3);
        [sendEventFall] sf = 3 & NE -> (sf'=4);
        [persistPersistenceFall] sf = 4 & NE -> (rPersistence) : (sf'=5) + (1-rPersistence) : (sf'=0);
        [persistPersistenceFall_return] sf = 5 & NE -> (sf'=6);
        [sendSituationFall] sf = 6 & NE -> (rBus) : (sf'=7) + (1-rBus) : (sf'=0);
        [sendSituationFall_return] sf = 7 & NE -> (sf'=7);
endmodule
module Persistence
        sr : [0..8] init 1;
        [failsr] sr = 0 -> (sr'=0);//Fail saate
        [persistPersistencePlsRt] sr = 1 & NE -> (sr'=2);
        [persistPersistenceOxy] sr = 1 & NE -> (sr'=2);
        [persistPersistenceTemp] sr = 1 & NE -> (sr'=2);
        [persistPersistencePos] sr = 1 & NE -> (sr'=2);
        [persistPersistenceFall] sr = 1 & NE -> (sr'=2);
        [fSqlite_Selection] sr = 2 & NE ->
                              (fSqlite) : (sr'=3) +    //fSqlite selected
                              (1 - fSqlite) : (sr'=5);//fSqlite deselected
        [persistSqlite] sr = 3 & NE -> (rSqlite) : (sr'=4) + (1-rSqlite) : (sr'=0);
```

```
        [persistSqlite_return] sr = 4 & NE -> (sr'=5);
        [fMem_Selection] sr = 5 & NE ->
                                (fMem) : (sr'=6) +      //fMem selected
                                (1 - fMem) : (sr'=8);//fMem deselected
        [persistMem] sr = 6 & NE -> (rMem) : (sr'=7) + (1-rMem) : (sr'=0);
        [persistMem_return] sr = 7 & NE -> (sr'=8);
        [persistPersistencePlsRt_return] sr = 8 & NE -> (rPlsRt) : (sr'=1) + (1-rPlsRt) : (sr'=0);
        [persistPersistenceOxy_return] sr = 8 & NE -> (rOxy) : (sr'=1) + (1-rOxy) : (sr'=0);
        [persistPersistenceTemp_return] sr = 8 & NE -> (rTemp) : (sr'=1) + (1-rTemp) : (sr'=0);
        [persistPersistencePos_return] sr = 8 & NE -> (rPos) : (sr'=1) + (1-rPos) : (sr'=0);
        [persistPersistenceFall_return] sr = 8 & NE -> (rFall) : (sr'=1) + (1-rFall) : (sr'=0);
endmodule
module SQLite
        sq : [0..2] init 1;
        [failsq] sq = 0 -> (sq'=0);//Fail saate
        [persistSqlite] sq = 1 & NE -> (sq'=2);
        [persistSqlite_return] sq =2 & NE -> (rPersistence) : (sq'=1) + (1-rPersistence) : (sq'=0);
endmodule
module Memory
        sm : [0..2] init 1;
        [failsm] sm = 0 -> (sm'=0);//Fail saate
        [persistMem] sm = 1 & NE -> (sm'=2);
        [persistMem_return] sm =2 & NE -> (rPersistence) : (sm'=1) + (1-rPersistence) : (sm'=0);
endmodule
module AdaptationManager
        si : [0..15] init 1;
        [failsi] si = 0 -> (si'=0);//Fail saate
        [sendEventAM] si = 1 & NE -> (si'=2);
        [getQosRequired] si = 2 & NE -> (rStateMachine) : (si'=3) + (1 - rStateMachine) : (si'=0);
        [getQosRequired_return] si =3 & NE -> (si'=4);
        [ntfyChngQosRequired] si = 4 & NE -> (rAM) : (si'=5) + (1 - rAM) : (si'=0);
        [sendEventAM_return] si = 5 & NE -> (rAM) : (si'=6) + (1 - rAM) : (si'=0);
        [start_Reconfiguration] si = 6 & NE -> (si'=7);
        [getQosRequiredAM] si = 7 & NE -> (rStateMachine) : (si'=8) + (1 - rStateMachine) : (si'=0);
        [getQosRequiredAM_return] si = 8 & NE -> (si'=9);
        [searchNewConfiguration] si = 9 & NE -> (rSPLManager) : (si'=10) + (1 - rSPLManager) : (si'=0);
        [searchNewConfiguration_return] si = 10 & NE -> (si'=11);
        [calculateQos] si = 11 & NE -> (rQosManager) : (si'=12) + (1 - rQosManager) : (si'=0);
        [calculateQos_return] si = 12 & NE -> (si'=13);
        [configure] si = 13 & NE -> (rConfigurator) : (si'=14) + (1 - rConfigurator) : (si'=0);
        [configure_return] si = 14 & NE -> (si'=15);
        [end_Reconfiguration] si = 15 & NE -> (si'=15);//END
endmodule
module StateMachineManager
        sg : [0..5] init 1;
        [failsg] sg = 0 -> (sg'=0);//Fail saate
        [getQosRequired] sg = 1 & NE -> (sg'=2);
        [getQosRequired_return] sg = 2 & NE -> (rAM) : (sg'=3) + (1 - rAM) : (sg'=0);
        [getQosRequiredAM] sg = 3 & NE -> (sg'=4);
        [getQosRequiredAM_return] sg = 4 & NE -> (rAM) : (sg'=5) + (1 - rAM) : (sg'=0);
endmodule
module SPLManager
        sx: [0..3] init 1;
        [failsx] sx = 0 -> (sx'=0);//Fail saate
        [searchNewConfiguration] sx = 1 & NE -> (sx'=2);
        [searchNewConfiguration_return] sx = 2 & NE -> (rAM) : (sx'=3) + (1 - rAM) : (sx'=0);
endmodule
module QoSManager
        sy: [0..3] init 1;
        [failsy] sy = 0 -> (sy'=0);//Fail saate
        [calculateQos] sy = 1 & NE -> (sy'=2);
        [calculateQos_return] sy = 2 & NE -> (rAM) : (sy'=3) + (1 - rAM) : (sy'=0);
endmodule
module Configurator
        sz: [0..3] init 1;
        [failsz] sz = 0 -> (sz'=0);//Fail saate
        [configure] sz = 1 & NE -> (sz'=2);
        [configure_return] sz = 2 & NE -> (rAM) : (sz'=3) + (1 - rAM) : (sz'=0);
endmodule
```

Listagem A.2: Expressão PCTL

```
P = ? [ true U (sb = 36 | si = 15) & NE ]
```

A.2.2 Composicional

Listagem A.3: Modelo PARAM - Nível de Atividades

```
dtmc
param double capture;
param double situation;
param double qosgoal1;
param double qosgoal2;
param double reconfiguration;
module VitalSignalMonitoring
        s : [0..5] init 1;
        [fail] s = 0 -> (s'=0);
        [SYSTEM_CAPTURE_VITAL_SIGNAL] s = 1 -> capture : (s'=2) + (1-capture) : (s'=0);
        [SYSTEM_IDENTIFIES_SITUATION] s = 2 -> situation : (s'=3) + (1-situation) : (s'=0);
        [CHANGE_QOS_GOAL] s = 3 -> qosgoal1 : (s'=4) + qosgoal2 : (s'=5) + (1-qosgoal1-qosgoal2) : (s'=0);
```

```
        [SYSTEM_RECONFIGURE] s = 4 -> reconfiguration : (s'=5) + (1-reconfiguration) : (s'=0);
        [END] s = 5 -> (s'=5);
endmodule
```

Listagem A.4: Expressão PCTL - Nível de Atividades

```
P = ? [ true U s = 5 ]
```

Listagem A.5: Modelo PARAM - Nível de Componentes (Atividade Capture Photo)

```
dtmc
//Features
param int fSSPO2;
param int fSTemp;
param int fSECG;
param int fSACC;
param int fOxy;
param int fTemp;
param int fPlsRt;
param int fPos;
param int fFall;
param int fMem;
param int fSqlite;
//Reliability
const double r = 0.999;
const double rBM = r;
const double rSECG = r;
const double rSSPO2 = r;
const double rSTemp = r;
const double rSAcc = r;
const double rBus = r;
const double rOxy = r;
const double rTemp = r;
const double rPlsRt = r;
const double rPos  = r;
const double rFall  = r;
const double rPersistence = r;
const double rSqlite = r;
const double rMem = r;
const double rAM = r;
const double rStateMachine = r;
const double rSPLManager = r;
const double rQosManager = r;
const double rConfigurator = r;
//Usage Profile
const double reconfigProfile = 0.5;
formula NE  = (s0!=0 & se!=0 & ss!=0 & st!=0 & sa!=0 & sb!=0);
module BluetoothManager
        s0 : [0..3] init 1;
        pECG : [0..1] init 0;
        pSPO2 : [0..1] init 0;
        pTEMP: [0..1] init 0;
        pACC: [0..1] init 0;
        [fails0] s0 = 0 -> (s0'=0);//Fail State
        [] s0 = 1 ->  (1-fSSPO2)*(fSECG)  *(1-fSTemp)*(1-fSACC) : (s0'=2)&(pECG'=1)
                + (fSSPO2)  *(1-fSECG)*(1-fSTemp)*(1-fSACC) : (s0'=2)&(pSPO2'=1)
                + (1-fSSPO2)*(1-fSECG)*(fSTemp)  *(1-fSACC) : (s0'=2)&(pTEMP'=1)
                + (fSSPO2)  *(fSECG)  *(1-fSTemp)*(1-fSACC) : (s0'=2)&(pECG'=1)&(pSPO2'=1)
                + (1-fSSPO2)*(fSECG)  *(fSTemp)  *(1-fSACC) : (s0'=2)&(pECG'=1)&(pTEMP'=1)
                + (fSSPO2)  *(1-fSECG)*(fSTemp)  *(1-fSACC) : (s0'=2)&(pSPO2'=1)&(pTEMP'=1)
                + (fSSPO2)  *(fSECG)  *(fSTemp)  *(1-fSACC) : (s0'=2)&(pSPO2'=1)&(pTEMP'=1)&(pECG'=1)
                + (1-fSSPO2)*(1-fSECG)*(1-fSTemp)*(1-fSACC) : (s0'=0)
                + (1-fSSPO2)*(fSECG)  *(1-fSTemp)*(fSACC) : (s0'=2)&(pECG'=1)&(pACC'=1)
                + (fSSPO2)  *(1-fSECG)*(1-fSTemp)*(fSACC) : (s0'=2)&(pSPO2'=1)&(pACC'=1)
                + (1-fSSPO2)*(1-fSECG)*(fSTemp)  *(fSACC) : (s0'=2)&(pTEMP'=1)&(pACC'=1)
                + (fSSPO2)  *(fSECG)  *(1-fSTemp)*(fSACC) : (s0'=2)&(pECG'=1)&(pSPO2'=1)&(pACC'=1)
                + (1-fSSPO2)*(fSECG)  *(fSTemp)  *(fSACC) : (s0'=2)&(pECG'=1)&(pTEMP'=1)&(pACC'=1)
                + (fSSPO2)  *(1-fSECG)*(fSTemp)  *(fSACC) : (s0'=2)&(pSPO2'=1)&(pTEMP'=1)&(pACC'=1)
                + (fSSPO2)  *(fSECG)  *(fSTemp)  *(fSACC) : (s0'=2)&(pSPO2'=1)&(pTEMP'=1)&(pECG'=1)&(pACC'=1)
                + (1-fSSPO2)*(1-fSECG)*(1-fSTemp)*(fSACC) : (s0'=2)&(pACC'=1);
        [sendPacket] s0 = 2 & pECG=1 & NE -> (rSECG) : (s0'=3) + (1 - rSECG) : (s0'=0);
        [sendPacket] s0 = 2 & pSPO2=1 & NE-> (rSSPO2) : (s0'=3) + (1 - rSSPO2) : (s0'=0);
        [sendPacket] s0 = 2 & pTEMP=1 & NE-> (rSTemp) : (s0'=3) + (1 - rSTemp) : (s0'=0);
        [sendPacket] s0 = 2 & pACC =1 & NE-> (rSAcc) : (s0'=3) + (1 - rSAcc) : (s0'=0);
endmodule
module ECG
        se : [0..3] init 1;
        se1 : [0..1] init 0;
        [failse] se = 0 -> (se'=0);//Fail State
        [sendPacket] se = 1 & NE-> (se'=2);
        [sendSituationSe] se = 2 & NE & (sa1 = 0 & st1 = 0 & ss1 = 0 & se1 = 0)  -> (rBus) : (se'=2) & (se1'=1)
        + (1 - rBus) : (se'=0);
        [sendPacket_returnSe] se = 2 & NE ->(rSECG) : (se'=3) + (1 - rSECG) : (se'=0);
endmodule
module SPO2
        ss : [0..3] init 1;
        ss1 : [0..1] init 0;
        [failss] ss = 0 -> (ss'=0);//Fail State
        [sendPacket] ss = 1 & NE -> (ss'=2);
        [sendSituationSs] ss = 2 & NE & (sa1 = 0 & st1 = 0 & ss1 = 0 & se1 = 0) -> (rBus) : (ss'=2) & (ss1'=1)
        + (1 - rBus) : (ss'=0);
        [sendPacket_returnSs] ss = 2 & NE -> (rSSPO2) : (ss'=3) + (1 - rSSPO2) : (ss'=0);
```

```
endmodule
module Temp
        st : [0..3] init 1;
        st1 : [0..1] init 0;
        [failst] st = 0 -> (st'=0);//Fail State
        [sendPacket] st = 1 & NE -> (st'=2);
        [sendSituationSt] st = 2 & NE & (sa1 = 0 & st1 = 0 & ss1 = 0 & se1 = 0)  -> (rBus) : (st'=2) & (st1'=1)
        + (1 - rBus) : (st'=0);
        [sendPacket_returnSt] st = 2 & NE ->(rSTemp) : (st'=3) + (1 - rSTemp) : (st'=0);
endmodule
module ACC
        sa : [0..3] init 1;
        sa1 : [0..1] init 0;
        [failsa] sa = 0 -> (sa'=0);//Fail saate
        [sendPacket] sa = 1 & NE -> (sa'=2);
        [sendSituationSa] sa = 2 & NE & (sa1 = 0 & st1 = 0 & ss1 = 0 & se1 = 0) -> (rBus) : (sa'=2) & (sa1'=1)
        + (1 - rBus) : (sa'=0);
        [sendPacket_returnSa] sa = 2 & NE -> (rSAcc) : (sa'=3) + (1 - rSAcc) : (sa'=0);
endmodule
module Bus
        sb : [0..37] init 1;
        [failsb] sb = 0 -> (sb'=0);//Fail saate
        [sendPacket_returnSa] sb = 1 & NE &  (sa1 = 1 | st1 = 1 | ss1 = 1 | se1 = 1) -> (sb'=2);
        [sendPacket_returnSt] sb = 1 & NE &  (sa1 = 1 | st1 = 1 | ss1 = 1 | se1 = 1) -> (sb'=2);
        [sendPacket_returnSs] sb = 1 & NE &  (sa1 = 1 | st1 = 1 | ss1 = 1 | se1 = 1) -> (sb'=2);
        [sendPacket_returnSe] sb = 1 & NE &  (sa1 = 1 | st1 = 1 | ss1 = 1 | se1 = 1) -> (sb'=2);
        [] sb = 2 & NE -> (sb'=2);
endmodule
```

Listagem A.6: Expressão PCTL - Nível de Componentes (Atividade Capture Photo)

```
P = ? [ true U sb = 2 & NE]
```

Listagem A.7: Modelo PARAM - Nível de Componentes (Atividade Capture Photo)

```
dtmc
//Features
param int fSSPO2;
param int fSTemp;
param int fSECG;
param int fSACC;
param int fOxy;
param int fTemp;
param int fPlsRt;
param int fPos;
param int fFall;
param int fMem;
param int fSqlite;
//Reliability
const double r = 0.999;
const double rBM = r;
const double rSECG = r;
const double rSSPO2 = r;
const double rSTemp = r;
const double rSAcc = r;
const double rBus = r;
const double rOxy = r;
const double rTemp = r;
const double rPlsRt = r;
const double rPos  = r;
const double rFall  = r;
const double rPersistence = r;
const double rSqlite = r;
const double rMem = r;
const double rAM = r;
const double rStateMachine = r;
const double rSPLManager = r;
const double rQosManager = r;
const double rConfigurator = r;
//Usage Profile
const double reconfigProfile = 0.5;
formula NE  = (sb!=0 & sp!=0 & so!=0 & su!=0 & sn!=0 & sf!=0 & sr!=0 & sq!=0 & sm!=0);
module Bus
        sb : [0..37] init 2;
        [failsb] sb = 0 -> (sb'=0);//Fail saate
        [fPlsRt_Selection] sb = 2 &  NE ->
                                (fPlsRt) : (sb'=3) +    //fPlsRt selected
                                (1 - fPlsRt) : (sb'=8);//fPlsRt deselected
        [registerPlsRt] sb = 3 & NE ->  (sb'=4);
        [registerPlsRt_return] sb = 4 & NE ->  (rPlsRt) : (sb'=5) + (1-rPlsRt) : (sb'=0);
        [sendEventPslRt] sb = 5 & NE ->  (rPlsRt) : (sb'=6) + (1-rPlsRt) : (sb'=0);
        [sendSituationPslRt] sb = 6 & NE ->  (sb'=7);
        [sendSituationPslRt_return] sb = 7 & NE ->  (rPlsRt) : (sb'=8) + (1 - rPlsRt) : (sb'=0);
        [fOxy_Selection] sb = 8 & NE ->
                                (fOxy) : (sb'=9) +      //fOxy selected
                                (1 - fOxy) : (sb'=14);//fOxy deselected
        [registerOxy] sb = 9 & NE ->  (sb'=10);
        [registerOxy_return] sb = 10 & NE ->  (rOxy) : (sb'=11) + (1-rOxy) : (sb'=0);
        [sendEventOxy] sb = 11 & NE ->  (rOxy) : (sb'=12) + (1-rOxy) : (sb'=0);
        [sendSituationOxy] sb = 12 & NE ->  (sb'=13);
        [sendSituationOxy_return] sb = 13 & NE ->  (rOxy) : (sb'=14) + (1 - rOxy) : (sb'=0);
        [fTemp_Selection] sb = 14 & NE ->
```

```
                              (fTemp) : (sb'=15) +     //fTemp selected
                              (1 - fTemp) : (sb'=20);//fTemp deselected
        [registerTemp] sb = 15 & NE ->  (sb'=16);
        [registerTemp_return] sb = 16 & NE ->  (rTemp) : (sb'=17) + (1-rTemp) : (sb'=0);
        [sendEventTemp] sb = 17 & NE ->  (rTemp) : (sb'=18) + (1-rTemp) : (sb'=0);
        [sendSituationTemp] sb = 18 & NE ->  (sb'=19);
        [sendSituationTemp_return] sb = 19 & NE ->  (rTemp) : (sb'=20) + (1 - rTemp) : (sb'=0);
        [fPos_Selection] sb = 20 & NE ->
                              (fPos) : (sb'=21) +      //fPos selected
                              (1 - fPos) : (sb'=26);//fPos deselected
        [registerPos] sb = 21 & NE ->  (sb'=22);
        [registerPos_return] sb = 22 & NE ->  (rPos) : (sb'=23) + (1-rPos) : (sb'=0);
        [sendEventPos] sb = 23 & NE ->  (rPos) : (sb'=24) + (1-rPos) : (sb'=0);
        [sendSituationPos] sb = 24 & NE ->  (sb'=25);
        [sendSituationPos_return] sb = 25 & NE ->  (rPos) : (sb'=26) + (1 - rPos) : (sb'=0);
        [fFall_Selection] sb = 26 & NE ->
                              (fFall) : (sb'=27) +     //fFall selected
                              (1 - fFall) : (sb'=32);//fFall deselected
        [registerFall] sb = 27 & NE ->  (sb'=28);
        [registerFall_return] sb = 28 & NE ->  (rFall) : (sb'=29) + (1-rFall) : (sb'=0);
        [sendEventFall] sb = 29 & NE ->  (rFall) : (sb'=30) + (1-rFall) : (sb'=0);
        [sendSituationFall] sb = 30 & NE ->  (sb'=31);
        [sendSituationFall_return] sb = 31 & NE ->  (rFall) : (sb'=32) + (1 - rFall) : (sb'=0);
        [END] sb = 32 & NE ->  (sb'=32);//END
endmodule
module PulseRate
        sp : [0..7] init 1;
        [failsp] sp = 0 -> (sp'=0);//Fail saate
        [registerPlsRt] sp = 1 & NE ->  (rBus) : (sp'=2) + (1-rBus) : (sp'=0);
        [registerPlsRt_return] sp = 2 & NE ->  (sp'=3);
        [sendEventPslRt] sp = 3 & NE ->  (sp'=4);
        [persistPersistencePlsRt] sp = 4 & NE ->  (rPersistence) : (sp'=5)  + (1-rPersistence) : (sp'=0);
        [persistPersistencePlsRt_return] sp = 5 & NE ->  (sp'=6);
        [sendSituationPslRt] sp = 6 & NE ->  (rBus) : (sp'=7) + (1-rBus) : (sp'=0);
        [sendSituationPslRt_return] sp = 7 & NE ->  (sp'=7);
endmodule
module Oxygenation
        so : [0..7] init 1;
        [failso] so = 0 -> (so'=0);//Fail saate
        [registerOxy] so = 1 & NE ->  (rBus) : (so'=2) + (1-rBus) : (so'=0);
        [registerOxy_return] so = 2 & NE ->  (so'=3);
        [sendEventOxy] so = 3 & NE ->  (so'=4);
        [persistPersistenceOxy] so = 4 & NE ->  (rPersistence) : (so'=5)  + (1-rPersistence) : (so'=0);
        [persistPersistenceOxy_return] so = 5 & NE ->  (so'=6);
        [sendSituationOxy] so = 6 & NE ->  (rBus) : (so'=7) + (1-rBus) : (so'=0);
        [sendSituationOxy_return] so = 7 & NE ->  (so'=7);
endmodule
module Temperature
        su : [0..7] init 1;
        [failsu] su = 0 -> (su'=0);//Fail suate
        [registerTemp] su = 1 & NE ->  (rBus) : (su'=2) + (1-rBus) : (su'=0);
        [registerTemp_return] su = 2 & NE ->  (su'=3);
        [sendEventTemp] su = 3 & NE ->  (su'=4);
        [persistPersistenceTemp] su = 4 & NE ->  (rPersistence) : (su'=5)  + (1-rPersistence) : (su'=0);
        [persistPersistenceTemp_return] su = 5 & NE ->  (su'=6);
        [sendSituationTemp] su = 6 & NE ->  (rBus) : (su'=7) + (1-rBus) : (su'=0);
        [sendSituationTemp_return] su = 7 & NE ->  (su'=7);
endmodule
module Position
        sn : [0..7] init 1;
        [failsn] sn = 0 -> (sn'=0);//Fail snate
        [registerPos] sn = 1 & NE ->  (rBus) : (sn'=2) + (1-rBus) : (sn'=0);
        [registerPos_return] sn = 2 & NE ->  (sn'=3);
        [sendEventPos] sn = 3 & NE ->  (sn'=4);
        [persistPersistencePos] sn = 4 & NE ->  (rPersistence) : (sn'=5)  + (1-rPersistence) : (sn'=0);
        [persistPersistencePos_return] sn = 5 & NE ->  (sn'=6);
        [sendSituationPos] sn = 6 & NE ->  (rBus) : (sn'=7) + (1-rBus) : (sn'=0);
        [sendSituationPos_return] sn = 7 & NE ->  (sn'=7);
endmodule
module Fall
        sf : [0..7] init 1;
        [failsf] sf = 0 -> (sf'=0);//Fail sfate
        [registerFall] sf = 1 & NE ->  (rBus) : (sf'=2) + (1-rBus) : (sf'=0);
        [registerFall_return] sf = 2 & NE ->  (sf'=3);
        [sendEventFall] sf = 3 & NE ->  (sf'=4);
        [persistPersistenceFall] sf = 4 & NE ->  (rPersistence) : (sf'=5)  + (1-rPersistence) : (sf'=0);
        [persistPersistenceFall_return] sf = 5 & NE ->  (sf'=6);
        [sendSituationFall] sf = 6 & NE ->  (rBus) : (sf'=7) + (1-rBus) : (sf'=0);
        [sendSituationFall_return] sf = 7 & NE ->  (sf'=7);
endmodule
module Persistence
        sr  : [0..8] init 1;
        [failsr] sr = 0 -> (sr'=0);//Fail saate
        [persistPersistencePlsRt] sr = 1 & NE ->  (sr'=2);
        [persistPersistenceOxy] sr = 1 & NE ->  (sr'=2);
        [persistPersistenceTemp] sr = 1 & NE ->  (sr'=2);
        [persistPersistencePos] sr = 1 & NE ->  (sr'=2);
        [persistPersistenceFall] sr = 1 & NE ->  (sr'=2);
        [fSqlite_Selection] sr = 2 & NE ->
                              (fSqlite) : (sr'=3) +   //fSqlite selected
                              (1 - fSqlite) : (sr'=5);//fSqlite deselected
        [persistSqlite] sr = 3 & NE ->  (rSqlite) : (sr'=4)  + (1-rSqlite) : (sr'=0);
        [persistSqlite_return] sr = 4 & NE ->  (sr'=5);
        [fMem_Selection] sr = 5 & NE ->
                              (fMem) : (sr'=6) +      //fMem selected
```

```
                        (1 - fMem) : (sr'=8);//fMem deselected
        [persistMem] sr = 6 & NE ->  (rMem) : (sr'=7)  + (1-rMem) : (sr'=0);
        [persistMem_return] sr = 7 & NE ->  (sr'=8);
        [persistPersistencePlsRt_return] sr = 8 & NE ->  (rPlsRt) : (sr'=1) + (1-rPlsRt) : (sr'=0);
        [persistPersistenceOxy_return] sr = 8 & NE ->  (rOxy) : (sr'=1) + (1-rOxy) : (sr'=0);
        [persistPersistenceTemp_return] sr = 8 & NE ->  (rTemp) : (sr'=1) + (1-rTemp) : (sr'=0);
        [persistPersistencePos_return] sr = 8 & NE ->  (rPos) : (sr'=1) + (1-rPos) : (sr'=0);
        [persistPersistenceFall_return] sr = 8 & NE ->  (rFall) : (sr'=1) + (1-rFall) : (sr'=0);
endmodule
module SQLite
        sq  : [0..2] init 1;
        [failsq] sq = 0 -> (sq'=0);//Fail saate
        [persistSqlite] sq = 1 & NE ->  (sq'=2);
        [persistSqlite_return] sq =2 & NE ->   (rPersistence) : (sq'=1)  + (1-rPersistence) : (sq'=0);
endmodule
module Memory
        sm  : [0..2] init 1;
        [failsm] sm = 0 -> (sm'=0);//Fail saate
        [persistMem] sm = 1 & NE ->  (sm'=2);
        [persistMem_return] sm =2 & NE ->   (rPersistence) : (sm'=1)  + (1-rPersistence) : (sm'=0);
endmodule
```

Listagem A.8: Expressão PCTL - Nível de Componentes (Atividade Capture Photo)

```
P = ? [ true U sb = 32 & NE ]
```

Listagem A.9: Modelo PARAM - Nível de Componentes (Atividade Capture Photo)

```
dtmc
//Features
param int fSSPO2;
param int fSTemp;
param int fSECG;
param int fSACC;
param int fOxy;
param int fTemp;
param int fPlsRt;
param int fPos;
param int fFall;
param int fMem;
param int fSqlite;
//Reliability
const double r = 0.999;
const double rBM = r;
const double rSECG = r;
const double rSSPO2 = r;
const double rSTemp = r;
const double rSAcc = r;
const double rBus = r;
const double rOxy = r;
const double rTemp = r;
const double rPlsRt = r;
const double rPos  = r;
const double rFall  = r;
const double rPersistence = r;
const double rSqlite = r;
const double rMem = r;
const double rAM = r;
const double rStateMachine = r;
const double rSPLManager = r;
const double rQosManager = r;
const double rConfigurator = r;
//Usage Profile
const double reconfigProfile = 0.5;
formula NE  = (sb!=0 & si!=0 & sg!=0);
module Bus
        sb : [0..37] init 32;
        [failsb] sb = 0 -> (sb'=0);//Fail saate
        [start_QosGoal] sb = 32 & NE ->  (sb'=37);
        [sendEventAM] sb = 37 & NE ->  (rAM) : (sb'=33) + (1 - rAM) : (sb'=0);
        [sendEventAM_return] sb = 33 & NE ->  (sb'=34);
        [profile_Decision] sb = 34 & NE ->
                        (reconfigProfile) : (sb'=35) +   //Reconfiguration Triggered
                        (1 - reconfigProfile) : (sb'=36);//End execution
        [start_Reconfiguration] sb = 35 & NE ->  (sb'=35);
        [end_QosGoal] sb = 36 & NE ->  (sb'=36);//END
endmodule
module AdaptationManager
        si : [0..15] init 1;
        [failsi] si = 0 -> (si'=0);//Fail saate
        [sendEventAM] si = 1 & NE ->  (si'=2);
        [getQosRequired] si = 2 & NE ->  (rStateMachine) : (si'=3) + (1 - rStateMachine) : (si'=0);
        [getQosRequired_return] si =3 & NE ->  (si'=4);
        [ntfyChngQosRequired] si = 4 & NE ->  (rAM) : (si'=5) + (1 - rAM) : (si'=0);
        [sendEventAM_return] si = 5 & NE ->  (rAM) : (si'=6) + (1 - rAM) : (si'=0);
endmodule
module StateMachineManager
        sg : [0..5] init 1;
        [failsg] sg = 0 -> (sg'=0);//Fail saate
        [getQosRequired] sg = 1 & NE ->  (sg'=2);
        [getQosRequired_return] sg = 2 & NE ->  (rAM) : (sg'=3) + (1 - rAM) : (sg'=0);
endmodule
```

Listagem A.10: Expressão PCTL - Nível de Componentes (Atividade Capture Photo)

```
P = ? [true U sb = 35 & NE]
```

Listagem A.11: Expressão PCTL - Nível de Componentes (Atividade Capture Photo)

```
P = ? [true U sb = 36 & NE]
```

Listagem A.12: Modelo PARAM - Nível de Componentes (Atividade Capture Photo)

```
dtmc
//Features
param int fSSPO2;
param int fSTemp;
param int fSECG;
param int fSACC;
param int fOxy;
param int fTemp;
param int fPlsRt;
param int fPos;
param int fFall;
param int fMem;
param int fSqlite;
//Reliability
const double r = 0.999;
const double rBM = r;
const double rSECG = r;
const double rSSPO2 = r;
const double rSTemp = r;
const double rSAcc = r;
const double rBus = r;
const double rOxy = r;
const double rTemp = r;
const double rPlsRt = r;
const double rPos  = r;
const double rFall  = r;
const double rPersistence = r;
const double rSqlite = r;
const double rMem = r;
const double rAM = r;
const double rStateMachine = r;
const double rSPLManager = r;
const double rQosManager = r;
const double rConfigurator = r;
//Usage Profile
const double reconfigProfile = 0.5;
formula NE  = (si!=0 & sg!=0 & sx!=0 & sy!=0 & sz!=0);
module AdaptationManager
        si : [0..15] init 6;
        [failsi] si = 0 -> (si'=0);//Fail saate
        [start_Reconfiguration] si = 6 & NE ->  (si'=7);
        [getQosRequiredAM] si = 7 & NE ->  (rStateMachine) : (si'=8) + (1 - rStateMachine) : (si'=0);
        [getQosRequiredAM_return] si = 8 & NE ->  (si'=9);
        [searchNewConfiguration] si = 9 & NE ->  (rSPLManager) : (si'=10) + (1 - rSPLManager) : (si'=0);
        [searchNewConfiguration_return] si = 10 & NE ->  (si'=11);
        [calculateQos] si = 11 & NE ->  (rQosManager) : (si'=12) + (1 - rQosManager) : (si'=0);
        [calculateQos_return] si = 12 & NE ->  (si'=13);
        [configure] si = 13 & NE ->  (rConfigurator) : (si'=14) + (1 - rConfigurator) : (si'=0);
        [configure_return] si = 14 & NE ->  (si'=15);
        [end_Reconfiguration]  si = 15 & NE ->  (si'=15);//END
endmodule
module StateMachineManager
        sg : [0..5] init 3;
        [failsg] sg = 0 -> (sg'=0);//Fail saate
        [getQosRequiredAM] sg = 3 & NE ->  (sg'=4);
        [getQosRequiredAM_return] sg = 4 & NE ->  (rAM) : (sg'=5) + (1 - rAM) : (sg'=0);
endmodule
module SPLManager
        sx: [0..3] init 1;
        [failsx] sx = 0 -> (sx'=0);//Fail saate
        [searchNewConfiguration] sx = 1 & NE ->  (sx'=2);
        [searchNewConfiguration_return] sx = 2 & NE ->  (rAM) : (sx'=3) + (1 - rAM) : (sx'=0);
endmodule
module QoSManager
        sy: [0..3] init 1;
        [failsy] sy = 0 -> (sy'=0);//Fail saate
        [calculateQos] sy = 1 & NE ->  (sy'=2);
        [calculateQos_return] sy = 2 & NE ->  (rAM) : (sy'=3) + (1 - rAM) : (sy'=0);
endmodule
module Configurator
        sz: [0..3] init 1;
        [failsz] sz = 0 -> (sz'=0);//Fail saate
        [configure] sz = 1 & NE ->  (sz'=2);
        [configure_return] sz = 2 & NE ->  (rAM) : (sz'=3) + (1 - rAM) : (sz'=0);
endmodule
```

Listagem A.13: Expressão PCTL - Nível de Componentes (Atividade Capture Photo)

```
P = ? [ true U si = 15 & NE ]
```

A.3 Fórmulas

Fórmulas omitidas devido à seu tamanho textual. A fim de ilustração apresentamos algumas fórmulas menores obtidas a partir de modelos utilizados na abordagem composicional.

Listagem A.14: Fórmula Parcial da abordagem composicional (relativa ao modelo da Listagem A.5 verificado com o PCTL da Listagem A.6)

```
[fSECG, fSSP02, fSTemp, fSACC]
[[0, 1] [0, 1] [0, 1] [0, 1]]
  (-997002999*fSECG*fSSP02*fSTemp*fSACC +997002999*fSECG*fSSP02*fSTemp +997002999*fSECG*fSSP02*fSACC -997002999*fSECG*
      fSSP02 +997002999*fSECG*fSTemp*fSACC -997002999*fSECG*fSTemp -997002999*fSECG*fSACC +997002999*fSECG
      +997002999*fSSP02*fSTemp*fSACC -997002999*fSSP02*fSTemp -997002999*fSSP02*fSACC +997002999*fSSP02 -997002999*
      fSTemp*fSACC +997002999*fSTemp +997002999*fSACC) / (1000000000)
```

As demais fórmulas estão disponíveis online em:
`https://code.google.com/p/spl-parametric-model-checking/`.

Referências

Jae-Hee Ahn and Yo-Sub Han. Implementation of state elimination using heuristics. In *Proceedings of the 14th International Conference on Implementation and Application of Automata*, CIAA '09, pages 178–187, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02978-3.

Rajeev Alur and Thomas A. Henzinger. Reactive modules. In *FORMAL METHODS IN SYSTEM DESIGN*, pages 207–218. IEEE Computer Society Press, 1996.

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004. ISSN 1545-5971.

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN 026202649X, 9780262026499.

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

Andrea Bianco and Luca De Alfaro. Model checking of probabilistic and nondeterministic systems. pages 499–513. Springer-Verlag, 1995.

J.A. Bondy and U.S.R Murty. *Graph Theory*. Springer, 1st edition, 2008. ISBN 1846289696.

Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE*, pages 335–344. ACM, 2010. ISBN 978-1-60558-719-6.

Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *ICSE*, pages 321–330. ACM, 2011. ISBN 978-1-4503-0445-0.

Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. ISBN 978-0-201-70332-0.

Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Towards an incremental automata-based approach for software product-line model checking. In *SPLC (2)*, pages 74–81, 2012.

Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000. ISBN 0201309777. URL http://www.worldcat.org/isbn/0201309777.

Conrado Daws. Symbolic and parametric model checking of discrete-time markov chains. In *Proceedings of ICTAC*, pages 280–294. Springer-Verlag, 2005. ISBN 3-540-25304-1, 978-3-540-25304-4.

Keith Ellul, Bryan Krawetz, Jeffrey Shallit, and Ming-wei Wang. Regular expressions: new results and open problems. *J. Autom. Lang. Comb.*, 9(2-3):233–256, September 2004. ISSN 1430-189X.

Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 261–270, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368124. URL http://doi.acm.org/10.1145/1368088.1368124.

Rohit Gheyi, Tiago Massoni, and Paulo Borba. Algebraic laws for feature models. *J. UCS*, 14(21):3573–3591, 2008.

Carlo Ghezzi and Amir Sharifloo. Quantitative verification of non-functional requirements with uncertainty. In *Dependable Computer Systems*, pages 47–62. Springer Berlin / Heidelberg, 2011a.

Carlo Ghezzi and Amir Molzam Sharifloo. Verifying non-functional properties of software product lines: Towards an efficient approach using parametric model checking. In *SPLC*, 2011b.

Gregor Gramlich and Georg Schnitger. Minimizing nfa's and regular expressions. *J. Comput. Syst. Sci.*, 73(6):908–923, September 2007. ISSN 0022-0000.

Gert-Martin Greuel and Gerhard Pfister. *A Singular Introduction to Commutative Algebra*. Springer Publishing Company, Incorporated, 2nd edition, 2007. ISBN 3540735410, 9783540735410.

Charles M. Grinstead and Laurie J. Snell. *Grinstead and Snell's Introduction to Probability*. American Mathematical Society, version dated 4 july 2006 edition, 2006. URL www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/book.html.

Lars Grunske. Specification patterns for probabilistic quality properties. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 31–40, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368094. URL http://doi.acm.org/10.1145/1368088.1368094.

Ernst Moritz Hahn. Parametric Markov Model Analysis. Master's thesis, Saarland University, Germany, 2008.

Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. Param: A model checker for parametric markov models. In *CAV*, pages 660–664, 2010.

Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:102–111, 1994.

Tero Harju. *Lecture Notes on Graph Theory*. University of Turku, 2011.

Leah Hoffman. In search of dependable design. *Commun. ACM*, 51(7), July 2008. ISSN 0001-0782.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363.

Intel. Intel core i7-2600 desktop processor series. `http://download.intel.com/support/processors/corei7/sb/core_i7-2600_d.pdf`, 2012.

Gurp Jilles Van, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, WICSA '01, pages 45–, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1360-3. doi: http://dx.doi.org/10.1109/WICSA.2001.948406. URL `http://dx.doi.org/10.1109/WICSA.2001.948406`.

S. M. Kay. *Intuitive Probability and Random Processes Using MATLAB*. Springer-Verlag, New York, first edition, 2006.

Charles W. Krueger. Towards a taxonomy for software product lines. In *Software Product Family Engineering*, pages 323–331, 2003.

M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In *SFM*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer, 2007.

M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

Robyn R. Lutz. Software engineering for safety: A roadmap. In *THE FUTURE OF SOFTWARE ENGINEERING*, pages 213–226. ACM Press, 2000.

Object Management Group. Uml 2.0. `http://www.omg.org/spec/UML/2.0/`, 2009.

Oracle. The JavaTM Virtual Machine Specification Java SE, may 2011. URL `docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf`.

Susan H. Rodger. JFLAP, June 2012. URL `www.jflap.org/`.

Genaína Nunes Rodrigues, Vander Alves, Renato Silveira, and Luiz A. Laranjeira. Dependability analysis in the ambient assisted living domain: An exploratory case study. *J. Syst. Softw.*, 85(1):112–131, January 2012. ISSN 0164-1212. doi: 10.1016/j.jss.2011.07.037. URL `http://dx.doi.org/10.1016/j.jss.2011.07.037`.

Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract features in feature modeling. In *SPLC*, pages 191–200, 2011.

Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13, January 2004. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/1005561.1005563. URL `http://doi.acm.org/10.1145/1005561.1005563`.

Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. The pla model: On the combination of product-line analyses. In *International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, New York, NY, USA, January 2013. ACM. To appear.

Douglas B. West. *Introduction to Graph Theory (2nd Ed)*. Prentice Hall, August 2000.