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

Provendo a fórmula paramétrica composicional como um serviço no PiStarGODA-MDP

Gabriel Levi Gomes da Silva

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

Orientadora
Prof.a Dr.a Genaína Nunes Rodrigues

Brasília
2021

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

Provendo a fórmula paramétrica composicional como um serviço no PiStarGODA-MDP

Gabriel Levi Gomes da Silva

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

Prof.a Dr.a Genaína Nunes Rodrigues (Orientadora)
CIC/UnB

Prof. Dr. Vander Ramos Alves
CIC/UnB

Prof. Dr. André Luiz Peron Martins Lanna
FGA/UnB

Prof. Dr. Marcelo Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 21 de maio de 2021

Dedicatória

Dedico este trabalho aos meus pais, Erenice e Geraldo, que me apoiaram em todos os momentos.

Agradecimentos

Agradeço aos meus pais, Erenice e Geraldo, pelo apoio, e também aos meus irmãos Júlia e Diogo. Agradeço à minha orientadora Prof.a Dr.a Genaína por ter me aceitado como orientando e pela ajuda durante meu PIBIC e este trabalho. Agradeço também ao Eric pela ajuda, dúvidas respondidas e revisões neste trabalho, ao Léo por fazer comigo todas as matérias possíveis, e ao Ricardo pela ajuda durante o PIBIC e após ele. Agradeço muito à Luiza que me ajudou muito em muitas coisas. Agradeço aos meus amigos que me acompanharam durante toda a graduação e ao meu gato, Perseu, que insistia em brincar comigo enquanto eu escrevia.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

O *framework* PiStarGODA-MDP é utilizado para realizar a análise de dependabilidade de sistemas por meio da modelagem utilizando modelos orientados a objetivos. Após realizar a verificação do modelo, o PiStarGODA-MDP gera fórmulas paramétricas para confiabilidade e custo, que podem ser utilizadas em tempo de execução. Estas fórmulas são atômicas e o seu uso em tempo de execução é limitado, uma vez que alterá-las dinamicamente não é trivial. Para solucionar isto, este trabalho apresenta a fórmula paramétrica em uma estrutura composicional em JSON, por meio da qual é possível disponibilizar serviços REST utilizando rotas HTTP implementadas no *framework* PiStarGODA-MDP. Assim, a fórmula paramétrica passa a ser um objeto dinâmico, permitindo consultas e edições na estrutura da mesma. Por fim, um estudo de caso foi realizado na BSN, visando servir como prova de conceito para validar a solução apresentada e sua implementação.

Palavras-chave: *Modelagem orientada a objetivos*, Pistar-GODA, fórmula paramétrica, análise quantitativa, composicionalidade.

Abstract

The PiStarGODA-MDP is a framework used to perform a dependability analysis through modeling using goal models. After performing the model verification, PiStarGODA-MDP generates parametric formulas for reliability and cost, which can be used at runtime. These formulas are atomic and their use at runtime is limited, since handling them is not trivial. To solve this, this work presents the parametric formula in a compositional structure in JSON, through which it is possible to make REST services available using HTTP routes implemented in the PiStarGODA-MDP framework. Thus, the parametric formula becomes a dynamic object, allowing queries and updates in its structure. Finally, the BSN system was used as a case study, qualified as a proof-of-concept to validate the presented solution and its implementation.

Keywords: Goal-oriented modelling, GODA, parametric formula, quantitative analysis, compositionality.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

O *Goal Oriented Dependability Analysis* (GODA) é um *framework* de análise de dependabilidade que une o poder do *Contextual Goal Model* (CGM), do *Runtime Goal Model* (RGM) e da *Probabilistic Model Checking* (PMC) para prover uma verificação formal de requisitos por meio do *model checking* [1]. Implementado inicialmente como um *plugin* para a IDE Eclipse, o GODA foi estendido e fornecido como serviço por Bergmann [2], que uniu a ferramenta PiStar [5] ao GODA, removendo sua dependência da IDE Eclipse e possibilitando o *deploy standalone*.

Após o desacoplamento do GODA da IDE Eclipse, o trabalho de Solano [3] estendeu o PiStarGODA, gerando a versão PiStarGODA-MDP, fornecendo modelos verificáveis que podem ser utilizados em tempo de modelagem e em tempo de execução de modelos não determinísticos MDP (*Markov Decision Process*). Com o trabalho de Solano, o PiStarGODA-MDP fornece uma fórmula paramétrica para custo e confiabilidade gerada a partir da modelagem orientada a objetivos. Com isso, sistemas auto adaptativos podem utilizar as fórmulas em tempo de execução, após modelagem no PiStarGODA-MDP, para cálculo de custo e confiabilidade, utilizando os valores como parâmetro para auto-adaptação.

Com a possibilidade de se utilizar a fórmula paramétrica de custo e confiabilidade, gerada a partir do PiStarGODA-MDP, em tempo de execução, surgem cenários onde pode ser necessário o manuseio da mesma de forma a modifica-lá, validá-la ou até mesmo consultar partes da mesma. Assim, é interessante tornar a estrutura da fórmula paramétrica composicional e fornecer modos de realizar consultas e edições na mesma, extrapolando o seu uso como fórmula estática, uma vez que sistemas auto-adaptativos são, por essência, dinâmicos.

Com a estrutura composicional, é possível estender o PiStarGODA-MDP com rotas, utilizando o protocolo HTTP, para consulta, criação e edição e assim, provendo a mesma como um serviço. As rotas podem ser utilizadas para validação da fórmula paramétrica, consultas em partes da fórmula e edições. Um cenário de uso de consultas em parte da fórmula é para o cálculo do custo de se realizar uma tarefa específica, com o objetivo de verificar se é possível realizar a mesma. Um outro cenário é a modificação da fórmula em tempo de execução, seja por inserção ou remoção de objetivos e tarefas do modelo, seja por mudança de contexto ou de modos de se atingir os objetivos em tempo de execução.

1.2 Problema

A fórmula paramétrica fornecida pelo PiStarGODA-MDP é estática e não há, atualmente, ferramentas para utilizar a fórmula paramétrica de um modo dinâmico. Uma vez que sistemas auto adaptativos operam em contextos dinâmicos, a estaticidade da fórmula demonstra ser uma limitação, tornando difícil o manuseio de partes da mesma e a realização de modificações. Dessa forma, uma evolução no sistema não é acompanhado por uma evolução na fórmula que o representa e assim, seria necessário uma nova modelagem e a geração de uma fórmula novamente. Além disso, a avaliação de tarefas ou objetivos fica limitada, pois a fórmula representa o sistema por completo e é difícil utilizar somente uma parte específica da fórmula para avaliar um objetivo ou tarefa. Para isso, o sistema teria que conhecer a fórmula correspondente de seus objetivos ou tarefas um a um.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo principal desse trabalho é estender o PiStarGODA-MDP e fornecer a fórmula paramétrica como um serviço utilizando uma estrutura composicional, de modo a facilitar seu uso em tempo de execução pelos sistemas modelados.

1.3.2 Objetivos específicos

De modo a prover a fórmula paramétrica como um serviço no PiStarGODA-MDP e validar a implementação, foram definidos os seguintes passos:

- Implementar um algoritmo para compor a fórmula paramétrica no *backend* do PiStarGODA-MDP;

- Implementar rotas utilizando o protocolo HTTP, com as operações de GET para consulta, PUT para edição e POST para criação de uma fórmula;
- Realizar uma prova de conceito utilizando um sistema auto-adaptativo e verificando que a implementação funciona como desejado e cumpre seu papel auxiliando no uso da fórmula paramétrica em tempo de execução.

1.4 Organização do Trabalho

Este trabalho se divide em mais quatro capítulos. O Capítulo 2 apresentará o referencial teórico utilizado como base para a realização do trabalho, trazendo conceitos sobre *goal model*, *probabilistic model checking* e o *framework* GODA e suas versões. Por sua vez, o Capítulo 3 contém a proposta e a implementação da solução para o problema descrito. O Capítulo 4 apresenta um estudo de caso como uma prova de conceito para as modificações apresentadas no Capítulo 3. Por fim, o Capítulo 5 conclui o trabalho.

Capítulo 2

Referencial teórico

2.1 Modelo de objetivos

Um modelo de objetivos, ou *goal model*, é uma estrutura hierárquica utilizada na engenharia de requisitos para representar os requisitos de um sistema. Este modelo é composto de objetivos, possuindo um objetivo raiz, que podem ser decompostos em outros objetivos ou em tarefas, as quais podem apenas ser decompostas em outras tarefas. Dessa forma, o modelo de objetivos provê um meio de analisar os requisitos de um sistema [1]. Uma metodologia que lida com modelos de objetivos é a Tropos [6], a qual define alguns conceitos-chave ao se trabalhar com estas estruturas, onde:

- **Atores** modelam uma entidade que tem objetivos estratégicos e intencionalidade dentro do sistema ou organização;
- **Objetivos** representam os interesses estratégicos do ator;
- **Tarefas** são ações ou um conjunto de ações que levam à realização de um dado objetivo.

De acordo com a metodologia Tropos, cada objetivo pode ser decomposto utilizando relações AND ou OR, utilizando apenas uma relação por vez. Uma decomposição AND requer que todos os subobjetivos ou tarefas sejam realizadas. A decomposição OR, por sua vez, requer que um ou mais objetivos ou tarefas sejam completados. A Figura 2.1 exemplifica um modelo de objetivos simples, onde o objetivo principal G1 é decomposto em outros dois objetivos, G2 e G3, utilizando relações AND, explicitando que ambos devem ser realizados para que G1 seja atingido. Por sua vez, G2 é decomposto nas tarefas folhas (ou *leaf tasks*, em inglês) G2.T1 e G2.T2, utilizando decomposição OR, onde é necessário que apenas uma das tarefas seja completada. De forma semelhante, o objetivo G3 é decomposto nas *leaf tasks* G3.T1 e G3.T2 utilizando decomposições OR.

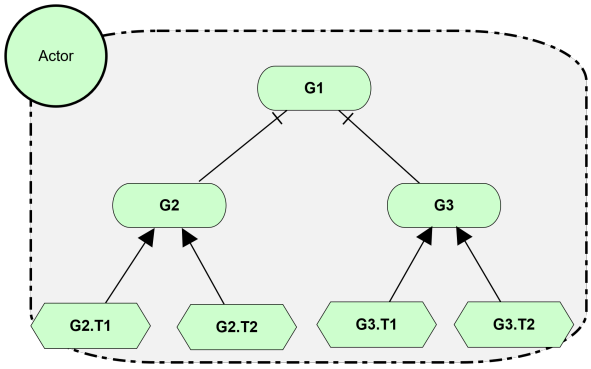

Figura 2.1: Exemplo de um modelo de objetivos.

2.1.1 *Contextual and runtime goal model*

O *contextual and runtime goal model* (CRGM) é um modelo de objetivos com informação sobre contexto, presentes no *contextual goal model* (CGM) [7] e de *runtime*, presentes no *runtime goal model* (RGM) [8]. Pode-se dizer então que o CRGM é um refinamento do conceito do CGM e do RGM [1]. De acordo com Ali et al. [7], as informações de contexto em um CGM devem ser avaliadas para cumprir os objetivos e podem influenciar os objetivos e as formas de atingi-los. O RGM, por sua vez, estende o modelo de objetivos provendo anotações e regras que são aplicadas em *runtime* [8], as quais são denominadas *runtime annotations*.

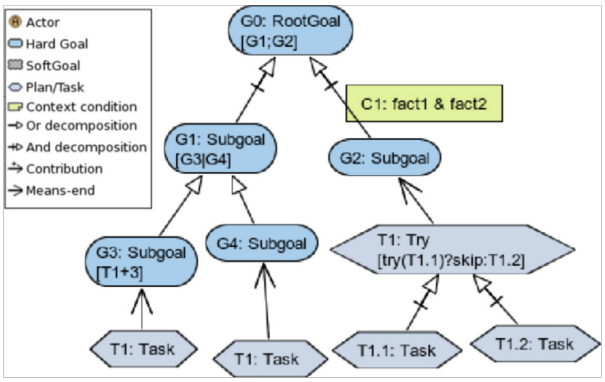

Figura 2.2: Exemplo de um CRGM [1].

A Figura 2.2 apresenta um exemplo de CRGM, onde temos informações de contexto em C1 e algumas *runtime annotations* em alguns objetivos e tarefas. As *runtime annotations* estão listadas na Tabela 2.1 abaixo. O objetivo G0 possui a anotação G1;G2, informando que G1 e G2 devem ser realizados sequencialmente. Já o objetivo G1 possui a anotação G3|G4, comunicando que G3 ou G4 devem ser realizados, porém não ambos. Por sua vez,

Tabela 2.1: Regras RGM, onde E, E1 e E2 representam *goals* ou *tasks* [2].

Expressão	**Significado**
AND(E1;E2)	Realização sequencial de E1 e E2
AND(E1#E2)	Realização em paralelo de E1 e E2
OR(E1;E2)	Realização sequencial de E1 ou E2, ou ambos
OR(E1#E2)	Realização em paralelo de E1 ou E2, ou ambos
E+n	E deve ser realizado n vezes, com n > 0
E#n	Realização em paralelo de n instâncias de E, com n > 0
E@n	Máximo de n-1 tentativas de realização de E, com n > 0
opt(E)	Realização de E é opcional
try(E)?E1:E2	Se E é executado, E1 deve ser executado; senão, E2
E1|E2	Realização alternativa de E1 ou E2, não ambos
skip	Sem ação. Útil em expressões ternárias condicionais

o objetivo G3 possui a anotação T1+3, explicitando que a tarefa G3.T1 deve ser realizada 3 vezes. Por fim, a tarefa G2.T1 possui a anotação try(T1.1)?skip:T1.2, e, portanto, possui o significado de que a tarefa G2.T1.1 deve tentar se executada, e se obter sucesso, faz um *skip*, se não obter sucesso, realiza a tarefa G2.T1.2.

2.2 *Probabilistic Model Checking*

Probabilistic Model Checking (PMC) é uma técnica de verificação de modelos que computa as probabilidades com as quais as propriedades de interesse são verificadas no sistema, em contraste às técnicas de checagem de modelos em que a corretude absoluta é verificada [9]. Dado um modelo de sistema e uma ou mais propriedades de dependabilidade, essa técnica pode ser utilizada para verificar a conformidade do modelo em cumprir tanto os requisitos funcionais quanto os requisitos não-funcionais do sistema [9]. Os modelos utilizados são anotados com probabilidades, sendo que *Discrete-Time Markov Chains* (DTMC) e *Markov Decision Processes* (MDP) [10] são comumente utilizados durante o processo de PMC [3].

O PRISM [11] é um *probabilistic model checker* que suporta DTMCs e MDPs como modelos, realizando a verificação dos mesmos e gerando PCTLs como resultado. Porém, a verificação pelo PRISM era limitada pela necessidade de se utilizar uma combinação única de variáveis inicializadas por vez em versões antigas, não permitindo a parametrização. Assim, a ferramenta PARAM foi desenvolvida como uma extensão ao PRISM para prover maior flexibilidade [9].

2.2.1 *Parametric Model Checking*

Parametric Model Checking é uma técnica formal para análise de cadeias de Markov cuja as a probabilidade das transições são especificadas como funções racionais [12]. As propriedades das cadeias de Markov analisadas pela *parametric model checking* são formalmente expressas como *probabilistic computation tree logic* (PCTL) [13] e tem como resultado uma expressão algébrica [14]. A expressão gerada pelo processo de *parametric model checking* corresponde a confiabilidade, desempenho ou alguma outra propriedade que expressa *quality-of-service* do sistema analisado [14]. Tal expressão pode, por exemplo, ser utilizada em tempo de execução por um sistema auto-adaptativo durante o processo de adaptação.

O PARAM [15] é uma extensão ao *model checker* PRISM, que adiciona a palavra reservada "param", permitindo que as variáveis que descrevem as probabilidades de transição sejam parametrizadas. Dessa forma, o PARAM faz o uso de *Parametric Markov Chains*, cuja as probabilidades transições são parametrizadas, para gerar uma expressão algébrica [15] usada para compor uma fórmula paramétrica.

2.3 *Goal Oriented Dependability Analysis*

O *Goal Oriented Dependability Analysis* (GODA) é um *framework* utilizado para fazer a análise de dependabilidade de um sistema que opera em contextos dinâmicos [1]. O GODA foi escrito em Java e proposto inicialmente como uma extensão do *plugin* TAOME4 para a IDE Eclipse, e leva em consideração aspectos de contexto e tempo de execução durante o processo de análise [1], utilizando as ferramentas PRISM [11] e PARAM [15] para *probabilistic model checking*.

O processo do GODA é ilustrado na Figura 2.3, no qual um modelo de objetivos é produzido e estendido com informações de contexto e *runtime*, produzindo então um CRGM. O CRGM, uma vez completo, é então automaticamente convertido em um DTMC. A partir do DTMC, são apresentadas as propriedades de dependabilidade como fórmulas PCTL e como resultado, o GODA fornece um modelo DTMC na linguagem PRISM e a fórmula paramétrica simbólica gerada pelo PARAM.

2.3.1 PiStarGODA

O PiStar [5] é uma ferramenta de *goal modelling open-source online*, que possibilita a modelagem no navegador, implementado utilizando a linguagem de programação Javascript. Visando transformar o GODA em um serviço, Bergmann [2] fez a remoção da dependência pelo TAOME4 e o substituiu pelo PiStar como *interface*, integrando o GODA como *bac-*

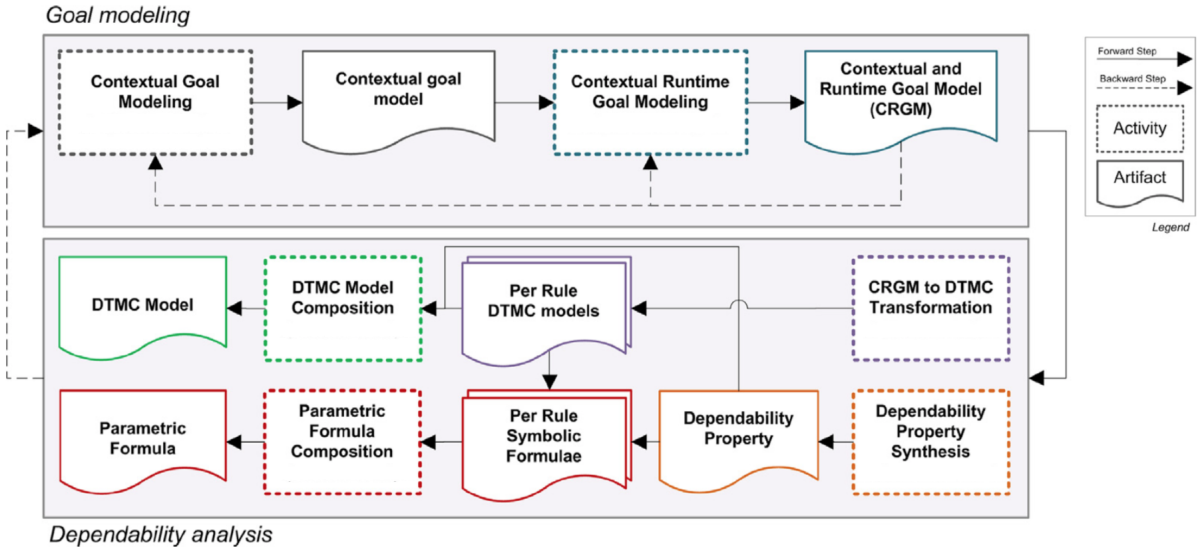

Figura 2.3: Processo do GODA [1].

kend e o PiStar como *frontend*, utilizando o *framework* Spring [16] para implementação de rotas REST no *controller*. Desse modo, o PiStarGODA não depende mais da ferramenta Eclipse. Dessa forma, Bergmann possibilitou que o GODA fosse provido como um serviço e que o PiStarGODA fosse disponibilizado como uma ferramenta web, com um *deploy* em um servidor.

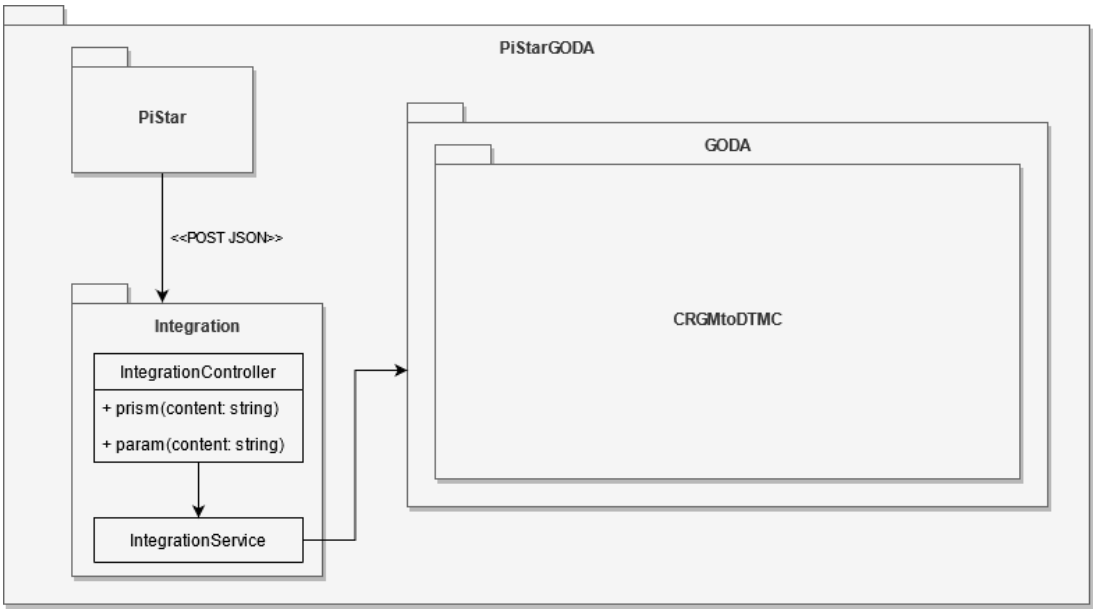

Figura 2.4: Arquitetura do PiStarGODA simplificada.

A Figura 2.4 contém a arquitetura feita por Bergmann. De acordo com a Figura 2.4, o PiStarGODA é composto por três módulos principais [2]:

- **GODA**: Código utilizado para transformação de modelos de objetivos em PRISM e PARAM;
- ***Integration***: Código que transforma os modelos PiStar em objetos Java e faz a integração com módulos GODA utilizando esses objetos;
- **PiStar**: Código fonte do PiStar, com algumas adaptações;

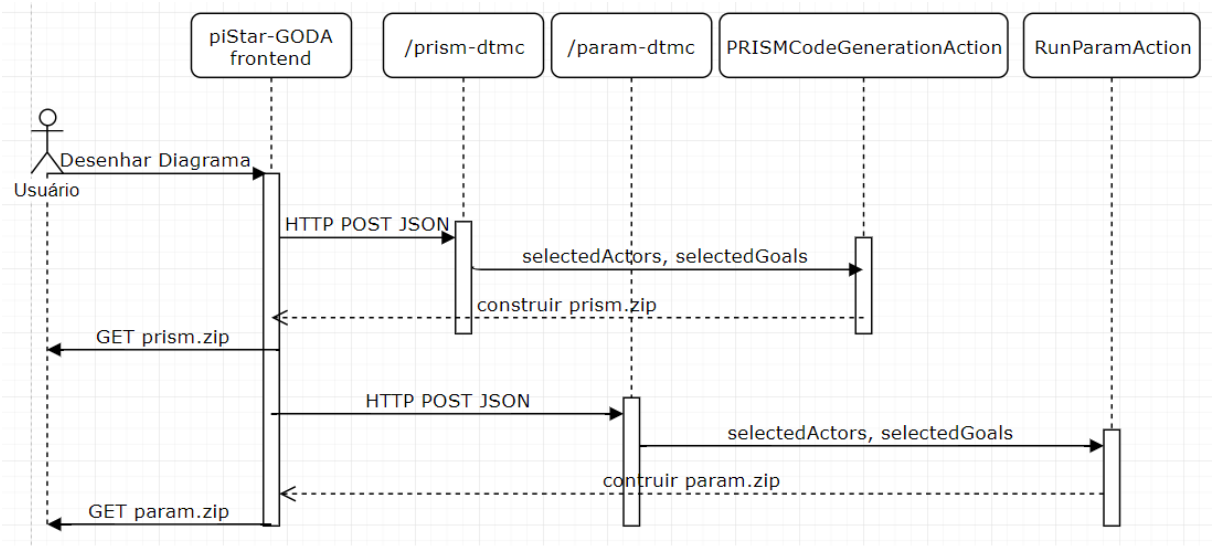

Figura 2.5: Diagrama de sequência do PiStarGODA [2].

A Figura 2.5 contém o diagrama de sequência do uso do PiStarGODA. Dessa forma, um CRGM é construído usando o PiStar no navegador e o mesmo é enviado ao *controller* por meio de um POST no formato de JSON. Esse CRGM é então transformado em um DTMC utilizando os processos do GODA. A partir do DTMC são feitas as análises de *probabilistic model checking* e *parametric model checking* por meio do PRISM e PARAM, seguindo o fluxo apresentado na Figura 2.3. Ao fim do processo, o PiStarGODA disponibiliza um arquivo para *download* contendo os resultados.

2.3.2 PiStarGODA-MDP

Segundo Mendonça et al.[1], a geração de uma fórmula paramétrica é essencial para uma análise eficaz de tempo e espaço em tempo de execução. Dessa forma, o trabalho de Solano [3] estende o PiStarGODA propondo o PiStarGODA-MDP, que utiliza MDPs e

a ferramenta de *parametric model checking* PARAM [15] para gerar uma fórmula paramétrica para confiabilidade e custo a partir da fórmula simbólica gerada pelo GODA anteriormente. A Figura 2.6 contém um resumo das fórmulas para cada tipo de nó de um CRGM.

Node type	**Reliability symbolic formula (P)**
AND $(N_1, ..., N_k)$	$\prod_{i=1}^{k} P_{n_i}$, where $k \in \mathbb{N}_{\geq 2}$
OR $(N_1, ..., N_k)$	$P_{n_1} + (1 - P_{n_1}) * P_{n_2} + ... + \left(\prod_{i=1}^{k-1}(1 - P_{n_i})\right) * P_{n_k}$, where $k \in \mathbb{N}_{\geq 2}$
$DM(N_1, ..., N_k)$	$C_{n_1}P_{n_1} + (1 - C_{n_1}P_{n_1}) * C_{n_2}P_{n_2} + ... + \left(\prod_{i=1}^{k-1}(1 - C_{n_i}P_{n_i})\right) * C_{n_k}P_{n_k}$, where $k \in \mathbb{N}_{\geq 2}$
Incompleteness (N_x)	$OPT_{n_X} * P_{n_X} - OPT_{n_X} + 1$

Node type	**Cost symbolic formula (W)**
AND $(N_1, ..., N_k)$	$P_{AND}(n_1, ..., n_k) * \sum_{i=1}^{k} W_{n_i}$, where $k \in \mathbb{N}_{\geq 2}$
OR $(N_1, ..., N_k)$	$-P_{OR}(n_1, ..., n_{k-1}) * W_{n_k} + P_{OR}(n_1, ..., n_k) * \sum_{i=1}^{k} W_{n_i}$, where $k \in \mathbb{N}_{\geq 2}$
$DM(N_1, ..., N_k)$	$-P_{DM}(n_1, ..., n_{k-1}) * C_{n_k}W_{n_k} + P_{DM}(n_1, ..., n_k) * \sum_{i=1}^{k} C_{n_i}W_{n_i}$, where $k \in \mathbb{N}_{\geq 2}$
Incompleteness (N_x)	$OPT_{n_X} * P_{n_X} * W_{n_X}$

Figura 2.6: Fórmula paramétrica para cada tipo de nó [3].

De acordo com a Figura 2.6, os parâmetros P, C e W representam, respectivamente, confiabilidade, contexto e custo. Por sua vez, o parâmetro N representa uma sub-árvore do CRGM. A anotação *DM* é uma sigla para *Decision Making*, que representa um comportamento não determinístico, proposto por Solano [3], e pode apenas ser inserido em nós não-folhas, e seu refinamento é composto pela decomposição OR dependente de contexto, exemplificado pela Figura 2.7, e evita que seja necessário enumerar todos os possíveis caminhos que surgem da variabilidade de contexto.

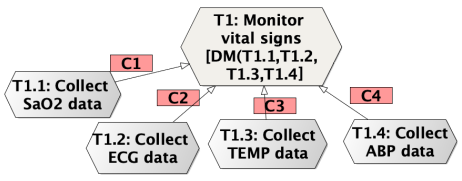

Figura 2.7: Exemplo de DM, modificado de Solano [3].

O processo para gerar uma fórmula paramétrica no PiStarGODA-MDP começa com a especificação e o desenho do CRGM na *interface*, por meio do PiStar. Após essa

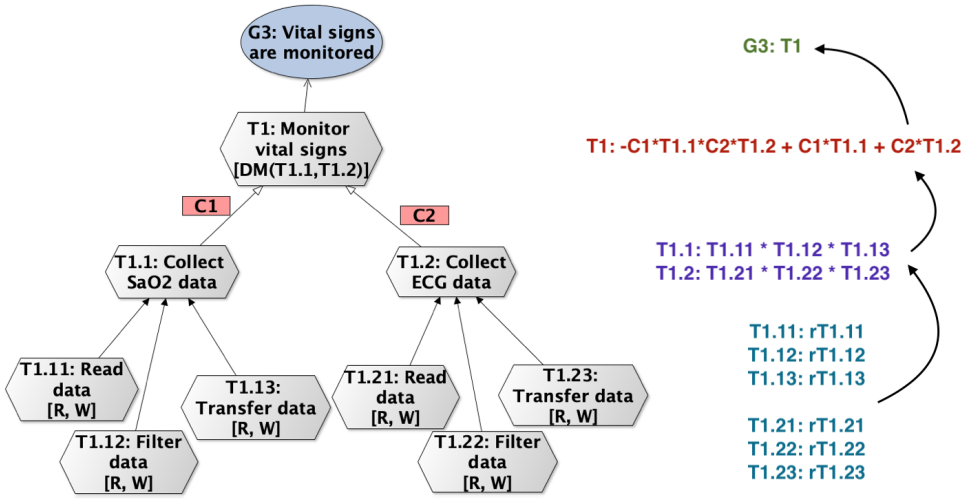

Figura 2.8: Exemplo de composição da fórmula para confiabilidade [3].

etapa, o CRGM é enviado ao *backend* e então é aplicado uma busca em profundidade no objeto que representa o CRGM, construindo um modelo PRISM [11] para cada *leaf task*. Então, o PARAM é utilizado para obter a uma expressão algébrica que, posteriormente, o GODA utilizará para gerar uma fórmula para confiabilidade e uma fórmula para custo de cada tarefa e, recursivamente, retornar os valores para serem compostos na fórmula total. Por fim, as fórmulas paramétricas de confiabilidade e custo são disponibilizadas em um arquivo texto para *download*. O processo de geração recursiva da fórmula paramétrica para confiabilidade é exemplificada na Figura 2.8, onde o algoritmo é aplicado em um modelo de objetivos de exemplo.

De acordo com a Figura 2.8, a confiabilidade do nó G3 é a confiabilidade de sua subárvore, que tem raiz T1. T1, por sua vez, possui uma anotação DM e sua fórmula segue a apresentada na Figura 2.6. Os nós T1.1 e T1.2 são refinados por meio de relações AND, e sua fórmula é composta pelo produtório das fórmulas das subárvores. Por fim, nas *leaf tasks*, as fórmulas são compostas pela confiabilidade e custo da tarefa em si. Dessa forma, a fórmula é gerada conforme o algoritmo desce na árvore, mas apenas simbolicamente, até chegar nas *leaf tasks*, onde então o algoritmo vai retornando na recursão substituindo os símbolos pelos valores de fato. No fim da execução, a fórmula gerada é dada em função dos valores das *leaf tasks*, das *runtime annotation* presentes nos objetivos e tarefas, e dos contextos dos mesmos.

Capítulo 3

Proposta

A fórmula paramétrica gerada pelo PiStarGODA-MDP é, essencialmente, estática, sendo disponibilizada como uma expressão algébrica única em um arquivo texto ao fim do processo. Portanto, qualquer tipo de processamento sobre a estrutura tem uma complexidade inerente de se trabalhar com objetos estáticos. Tal estrutura é planejada para a estimativa de custo e de confiabilidade em uma estrutura fixa, e não para acomodar modificações advindas de interações dinâmicas das estruturas que compõem a fórmula em si. Dessa forma, uma fórmula paramétrica composicional permite modificações com algoritmos mais simples e também uma avaliação mais fácil, além da utilização de apenas segmentos da mesma, possibilitando análises locais sobre os componentes do modelo de objetivos em tempo de execução.

Além de permitir modificações e usos mais interessantes em tempo de execução, tornar a fórmula paramétrica composicional utilizando o PiStarGODA-MDP também permite o uso da mesma como um serviço, via protocolo REST e seus métodos GET, POST e PUT. Portanto, ao prover a fórmula paramétrica como um serviço REST, torna-se possível não apenas acessar partes da fórmula paramétrica de forma composicional como também permitir que partes de sua estrutura sejam alteradas ou atualizadas em tempo de execução.

A Figura 3.1 contém a arquitetura da solução proposta. A classe *IntegrationController* deve ser estendida com as rotas GET, POST e PUT. No pacote *Integration*, será criada uma nova classe denominada *FormulaService* para realizar as operações de consulta, criação e edição de uma fórmula. Já no código do GODA, será incluído um novo método responsável por construir a árvore da fórmula paramétrica, denominado *generateFormulaTree*.

Sendo assim, este capítulo propõe tornar a fórmula paramétrica composicional, apresentando uma visão geral desse processo através de um exemplo com um modelo de objetivos simples. Também desenvolve a proposta de prover a fórmula como um serviço

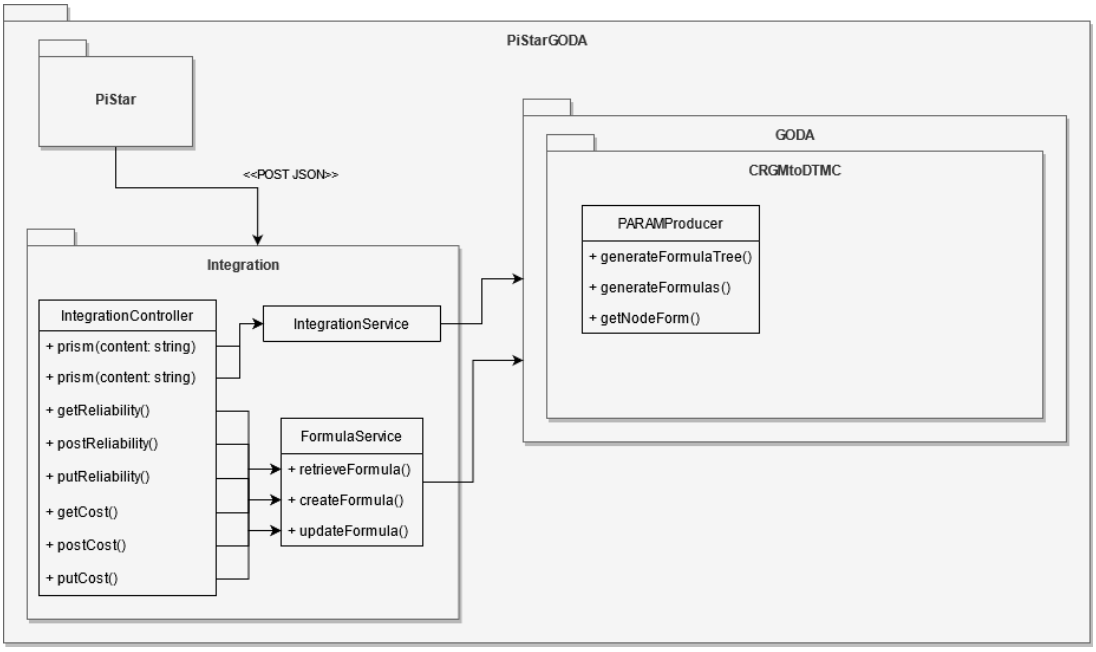

Figura 3.1: Arquitetura da solução proposta.

REST por meio das rotas GET, POST e PUT com um panorama dos processos utilizados. Por fim, a implementação no PiStarGODA-MDP, detalhando o algoritmo utilizado para compor a fórmula paramétrica e como o servidor salva o arquivo gerado. Além disso, descreve-se como a fórmula foi disponibilizada como um serviço, com as implementações das rotas para consulta, criação e modificação da mesma, utilizando a estrutura de árvore no formato JSON.

3.1 Fórmula paramétrica composicional

A proposta é transformar a fórmula paramétrica gerada pelo PiStarGODA-MDP em uma estrutura composicional, disposta em uma árvore e com isomorfismo com o modelo de objetivos original, de forma que seja possível validar a fórmula gerada, bem como utilização da mesma em tempo de execução, endereçando aspectos dinâmicos. Para isso, cada nó da fórmula paramétrica decomposta em uma estrutura de árvore JSON deverá conter informações necessárias para identificação e utilização da fórmula paramétrica, preservando e garantindo sua composicionalidade em tempo de execução.

A abordagem é ilustrada por meio da Figura 3.3, a qual exemplifica a fórmula paramétrica de confiabilidade em árvore, gerada a partir do modelo de objetivos presente na Figura 3.2. O nó G1 contém a fórmula completa, T1 e T2 contém a fórmula de suas

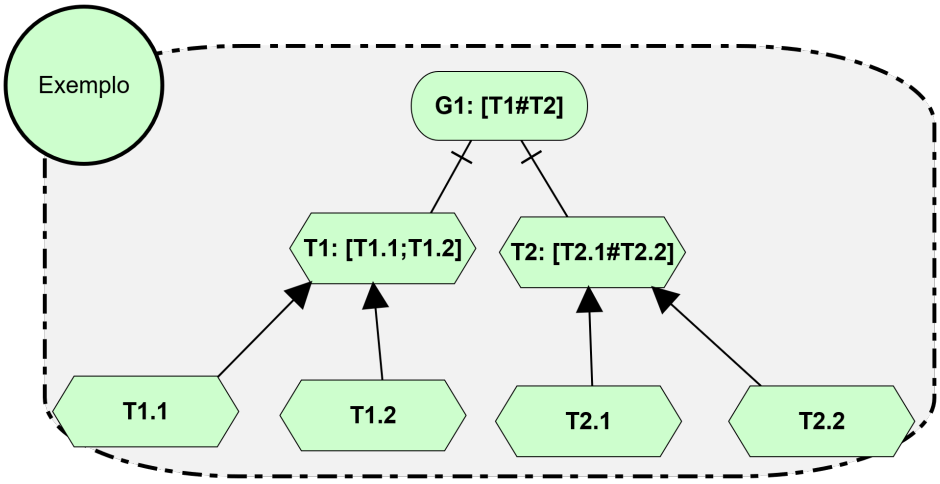

Figura 3.2: Exemplo de modelo de objetivos montado no piStarGODA-MDP.

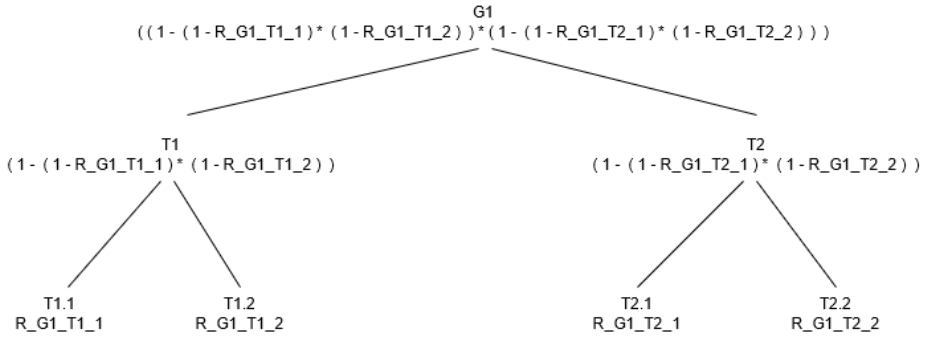

Figura 3.3: Fórmula paramétrica composicional do modelo de objetivos da Figura 3.2.

respectivas subárvores, compostas pelas tarefas-folhas. Por uma questão de simplificação, não há informação de contexto para as tarefas do modelo de objetivos.

3.2 Fórmula paramétrica como um serviço

A partir da composição da fórmula em árvore, a proposta é prover a fórmula como um serviço REST, com a implementação das rotas GET, POST e PUT para consulta, criação e edição da mesma, respectivamente. As rotas devem estar disponíveis sob um *endpoint* do PiStarGODA-MDP. Para tal, o PiStarGODA-MDP deve implementar a persistência da árvore gerada pelo usuário, retornando um identificador único para utilização dos serviços de GET e PUT.

3.2.1 GET

A proposta da rota de GET é implementar a consulta em uma árvore ou subárvore, utilizando o identificador gerado pelo PiStarGODA-MDP. A rota precisa fazer diferenciação entre custo e confiabilidade, recebendo como parâmetro o identificador da árvore e o nó retornado na consulta. Por ser somente de consulta, o serviço GET não pode fazer modificações na árvore, devendo apenas carregar a mesma para a memória, utilizar um algoritmo de percorrimento para buscar a subárvore e retorná-la.

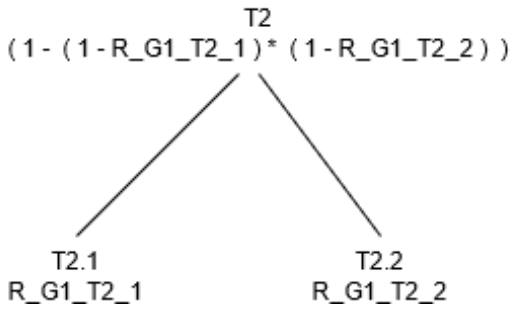

Figura 3.4: Fórmula retornada pelo GET como parâmetro T2.

Dessa forma, o objetivo da rota de GET é permitir que o usuário consulte a fórmula paramétrica em partes ou completa. Assim, a Figura 3.4 exemplifica o resultado de um GET realizado passando T2 como parâmetro. Portanto, o sistema modelado pode utilizar a fórmula de T2 retornada para fazer uma análise local e calcular o custo de se realizar a tarefa em tempo de execução, verificando se compensa a realização da mesma.

3.2.2 POST

A rota POST é a que possui implementação mais simples, recebendo a árvore por completo e o nome da mesma no corpo da requisição, e salvando-a utilizando a persistência do servidor. A rota deve retornar uma estrutura que contenha um identificador único gerado

no momento do salvamento e a árvore salva, se o processo foi realizado sem erros. Assim, a utilização dessa rota se dá em momentos em que se torna necessário a criação de uma nova fórmula.

3.2.3 PUT

A rota de PUT tem como objetivo permitir a edição de uma árvore já existente. Para tal, a rota precisa fazer diferenciação entre custo e confiabilidade e receber como parâmetro o identificador da árvore, o nó a ser editado e se a edição será ou não persistida. Seu retorno será a árvore editada por completa, com a fórmula completamente atualizada e correta. Para tal, é necessário carregar a árvore em memória e acessar o nó que a ser editado. Após isso, é necessário fazer as modificações necessárias e recalcular a fórmula para aquele nó. Então, utilizando um algoritmo de percorrimento e a noção de composicionalidade, o fragmento de fórmula antigo deve ser substituído pelo novo em toda a árvore. Dessa forma, a fórmula final estará correta.

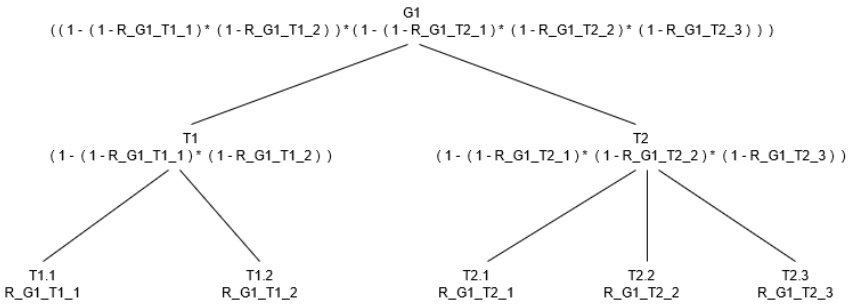

Figura 3.5: Fórmula retornada pelo PUT adicionando o nó T2.3.

A Figura 3.5 exemplifica o resultado esperado após utilizar a rota para adicionar um nó novo em T2. Assim, a fórmula é corrigida por completa adicionando um novo nó em toda a estrutura acima de T2, e sua fórmula é então utilizada para atualizar toda a árvore, fazendo substituições nas ocorrências da fórmula antiga. Dessa forma, a rota de PUT pode ser utilizada para retirar ou adicionar objetivos ou tarefas pelo sistema modelado, em tempo de execução. Assim, a rota adiciona mais dinamicidade à fórmula paramétrica e aproveita a composicionalidade para não gerar a fórmula inteira novamente, utilizando substituições.

3.3 Implementação

3.3.1 Compondo a fórmula paramétrica

A disponibilidade da fórmula paramétrica composicional depende da construção da mesma em formato de árvore. Para tal, o algoritmo utilizado se baseia em salvar cada etapa intermediária da fórmula paramétrica gerada pelo PiStarGODA-MDP. Portanto, uma nova classe foi criada no pacote *model*, denominada FormulaTreeNode, responsável por representar o nó da estrutura da fórmula em formato de árvore. A estrutura de dados em si é feita pelo encadeamento de FormulaTreeNode, apresentada no fragmento de código 3.1.

```
1 public class FormulaTreeNode {
2     public String id;
3     public String formula;
4     public String annotation;
5     public Const decomposition;
6     public boolean hasContext;
7     public List<FormulaTreeNode> subNodes = new LinkedList<>();
8 }
```

Listing 3.1: Classe que representa o nó da árvore

Devido ao fato de cada modelo de objetivos ser singular e não ter limites de tamanho, é necessário utilizar uma estrutura de dados que seja alocada dinamicamente, permitindo que cada subárvore tenha um número indeterminado de filhos. Dessa forma, a classe FormulaTreeNode é composta por um id, que guarda o nome do nó do modelo de objetivos, o campo formula, que guarda a fórmula composta da subárvore do qual este nó é raiz. Por sua vez, o campo annotation guarda a *runtime annotation* do nó. O campo decomposition contém a decomposição. Já a propriedade hasContext informa se o nó possui contexto ou não. Por fim, subNodes contém todos os filhos diretos do nó, no qual monta a relação entre os nós e a estrutura de árvore.

Para estruturar composicionalmente a fórmula, cada etapa de geração da mesma pelo GODA é salva em um FormulaTreeNode e recursivamente inserida na lista subNodes, construindo a relação entre os nós. Dessa forma, a raiz da árvore guarda a fórmula total e cada nó guarda a fórmula total da subárvore cujo esse nó é raiz. Assim, qualquer consulta em uma parte da fórmula que não seja na raiz depende da complexidade do algoritmo escolhido para percorrimento, sendo possível de ser realizado em O(N) no pior caso, onde N é o número de nós da árvore. O fragmento de código 3.2 é responsável por montar a árvore das fórmulas de confiabilidade e custo, sendo chamado após a geração das mesmas.

```
1  private FormulaTreeNode generateFormulaTree(RTContainer rootNode) {
2    FormulaTreeNode node = new FormulaTreeNode();
3    LinkedList<GoalContainer> decompGoal =
         removeDuplicates(rootNode.getDecompGoals());
4    LinkedList<PlanContainer> decompPlans =
         removeDuplicates(rootNode.getDecompPlans());
5
6    if (rootNode instanceof GoalContainer) {
7      node.id = rootNode.getClearUId();
8    } else {
9      node.id = rootNode.getClearElId();
10   }
11
12   node.formula = rootNode.getFormula();
13   node.annotation = rootNode.getRtRegex() != null ? rootNode.getRtRegex() : "";
14   node.decomposition = rootNode.getDecomposition();
15
16   for (GoalContainer subNode : decompGoal) {
17     FormulaTreeNode sn = generateFormulaTree(subNode);
18     sn.hasContext = node.formula.contains("CTX_" + sn.id + " *");
19     node.subNodes.add(sn);
20   }
21
22   for (PlanContainer subNode : decompPlans) {
23     FormulaTreeNode sn = generateFormulaTree(subNode);
24     sn.hasContext = node.formula.contains("CTX_" + sn.id + " *");
25     node.subNodes.add(sn);
26   }
27
28   return node;
29 }
```

Listing 3.2: Trecho de código que cria a árvore da fórmula paramétrica

A fórmula paramétrica é gerada pelo PiStarGODA-MDP de forma composicional, porém os valores da fórmula por nó durante a composição eram perdidos após a conclusão da etapa de geração. Dito isso, para construir a árvore com os campos necessários e com a estrutura correta, o método *generateFormulaTree* foi implementado na classe *PARAM-Producer* e é chamado após o fim da etapa de geração, passando como parâmetro a raiz

da árvore que representa o modelo de objetivos. Após a criação de um FormulaTreeNode vazio (linha 2), são chamados métodos responsáveis por retirar qualquer objetivo ou plano duplicados (linhas 3 e 4).

Adiante no código, é atribuído ao FormulaTreeNode o nome do objetivo ou tarefa do modelo de objetivos (linhas 6 a 10), permitindo que seja possível identificar o nó da árvore e sua relação com o modelo de objetivos. Então, a fórmula gerada anteriormente é atribuída ao FormulaTreeNode (linha 12), bem como a *runtime annotation* (linha 13) e sua decomposição (linha 14). O método é chamado recursivamente para cada subobjetivo ou tarefa, repetindo o mesmo processo (linhas 17 e 23). Para finalizar a recursão, o nó é retornado, e, após a verificação de contexto (linhas 18 e 24), adicionado na lista FormulaTreeNode.subNodes (linhas 19 e 25), representando as relações entre os nós da árvore. A árvore resultante possui isomorfismo com o modelo de objetivos utilizado, uma vez que o algoritmo utilizado para construí-la é uma busca em profundidade no modelo de objetivos.

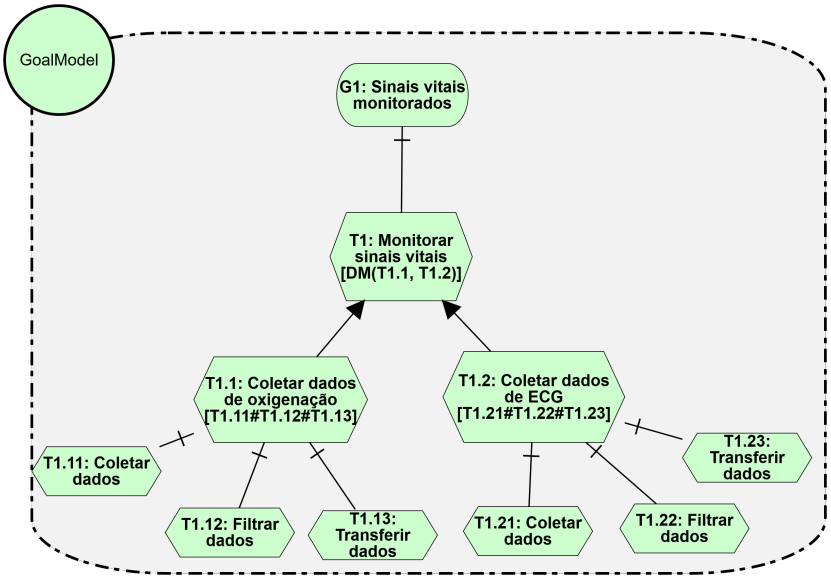

Figura 3.6: Exemplo de modelo de objetivos montado no piStarGODA-MDP.

```
1  {
2    "id" : "G1",
3    "formula" : "( 1 - ( 1 - ( R_G1_T1_11 * R_G1_T1_12 * R_G1_T1_13 ) ) * ( 1 - ( R_G1_T1_21 * R_G1_T1_22 *
         R_G1_T1_23 ) ) )",
4    "annotation" : "",
5    "decomposition" : "ME",
6    "hasContext" : false,
7    "subNodes" : [ {
8      "id" : "G1_T1",
9      "formula" : "( 1 - ( 1 - ( R_G1_T1_11 * R_G1_T1_12 * R_G1_T1_13 ) ) * ( 1 - ( R_G1_T1_21 * R_G1_T1_22
           * R_G1_T1_23 ) ) )",
```

```
    "annotation" : "DM(T1.1,T1.2)",
    "decomposition" : "OR",
    "hasContext" : false,
    "subNodes" : [ {
      "id" : "G1_T1_1",
      "formula" : "( R_G1_T1_11 * R_G1_T1_12 * R_G1_T1_13 )",
      "annotation" : "T1.11#T1.12#T1.13",
      "decomposition" : "AND",
      "hasContext" : false,
      "subNodes" : [ {
        "id" : "G1_T1_11",
        "formula" : "R_G1_T1_11",
        "annotation" : "",
        "decomposition" : "NONE",
        "hasContext" : false,
        "subNodes" : [ ]
      }, {
        "id" : "G1_T1_12",
        "formula" : "R_G1_T1_12",
        "annotation" : "",
        "decomposition" : "NONE",
        "hasContext" : false,
        "subNodes" : [ ]
      }, {
        "id" : "G1_T1_13",
        "formula" : "R_G1_T1_13",
        "annotation" : "",
        "decomposition" : "NONE",
        "hasContext" : false,
        "subNodes" : [ ]
      } ]
    }, {
      "id" : "G1_T1_2",
      "formula" : "( R_G1_T1_21 * R_G1_T1_22 * R_G1_T1_23 )",
      "annotation" : "T1.21#T1.22#T1.23",
      "decomposition" : "AND",
      "hasContext" : false,
      "subNodes" : [ {
        "id" : "G1_T1_21",
        "formula" : "R_G1_T1_21",
        "annotation" : "",
        "decomposition" : "NONE",
        "hasContext" : false,
        "subNodes" : [ ]
      }, {
        "id" : "G1_T1_22",
        "formula" : "R_G1_T1_22",
        "annotation" : "",
        "decomposition" : "NONE",
        "hasContext" : false,
        "subNodes" : [ ]
      }, {
        "id" : "G1_T1_23",
        "formula" : "R_G1_T1_23",
        "annotation" : "",
        "decomposition" : "NONE",
        "hasContext" : false,
```

```
67          "subNodes" : [ ]
68        } ]
69      } ]
70    } ]
71  }
```

Listing 3.3: Fórmula paramétrica em árvore gerada a partir do modelo de objetivos

A Figura 3.6 contém um modelo de objetivos modelado no PiStarGODA-MDP que foi utilizado para demonstrar a geração da fórmula paramétrica composicional em formato de árvore para confiabilidade, apresentada na Listagem 3.3. O arquivo JSON é obtido utilizando a biblioteca *ObjectMapper* do Java, que faz a conversão de um objeto para JSON. Com isso, um identificador único é obtido, utilizando o *timestamp* no momento da criação do arquivo JSON. Este identificador é concatenado ao nome dado ao modelo de objetivos pelo usuário e ao sufixo "_reliability"ou "_cost"para definir o nome do arquivo JSON no qual a árvore será salva.

Por fim, a árvore é escrita no arquivo JSON e salva pelo PiStarGODA-MDP. Dessa forma, a árvore fica disponível para consultas e modificações posteriores. Também é disponibilizada no arquivo compactado de *download* gerado as árvores de custo, confiabilidade e um arquivo contendo o identificador, que é usado posteriormente para utilizar os serviços de consulta e modificação fornecidos pelo PiStarGODA-MDP, utilizando as rotas de GET, POST e PUT.

3.3.2 Fórmula paramétrica como um serviço

A disponibilização da fórmula paramétrica composicional em formato JSON permite disponibilizar a mesma como um serviço, uma vez que JSON é o formato padrão para comunicação HTTP. Para tal, foi feita a implementação dos métodos GET, POST e PUT no IntegrationController do PiStarGODA-MDP, com rota para consultas, criação de uma nova árvore e modificação de uma árvore já existente. As rotas são públicas e sem autenticação, permitindo que qualquer usuário que possua o identificador de uma árvore consulte e modifique a mesma, além de criar novas árvores. Os métodos que as rotas utilizam foram implementados em uma nova classe chamada FormulaService, no módulo *Integration*.

GET

A rota de GET implementa a consulta na formula composicional e é disponibilizada sob o *endpoint* “/formula/<modo>?id=<identificador>&goal=<nó da árvore>”, onde “<modo>” descreve se a árvore é de custo ou confiabilidade, sendo as duas opções possíveis *“cost”* e *“reliability”*. O parâmetro “<identificador>”, por sua vez, é o identificador

gerado pelo PiStarGODA-MDP no momento da escrita do arquivo, e é usado para identificar o arquivo do qual a árvore será carregada para a memória. Por fim, o último parâmetro diz respeito ao nó da árvore que se deseja consultar, explorando a composicionalidade da fórmula e permitindo que qualquer parte da mesma seja obtida.

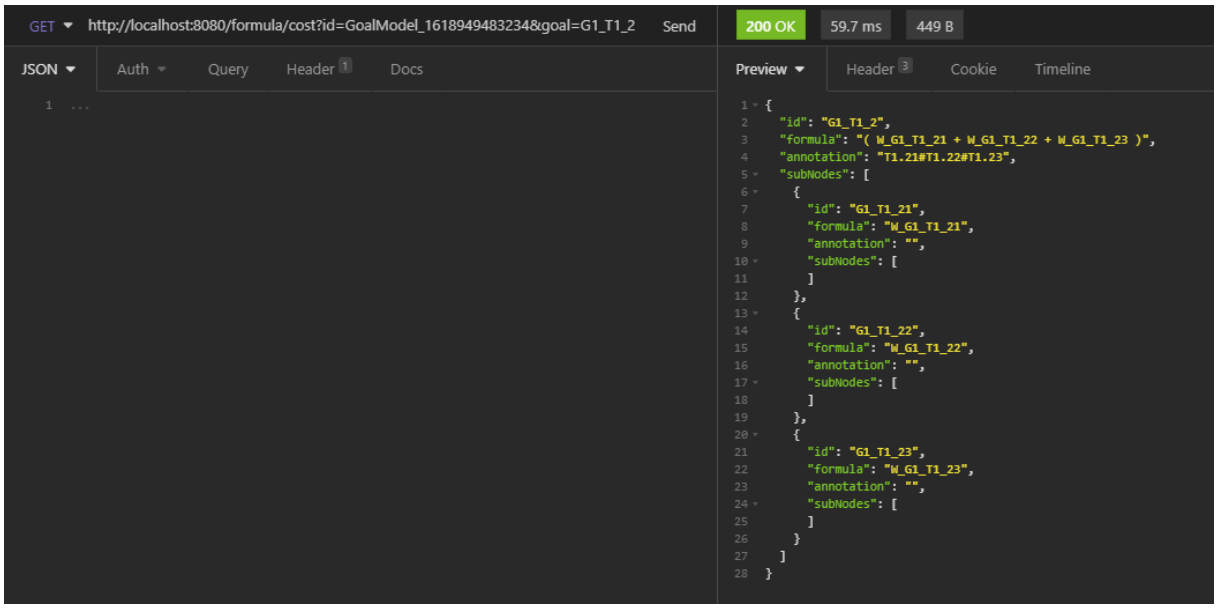

Figura 3.7: Requisição GET e retorno.

A Figura 3.7 exemplifica uma requisição GET ao servidor do PiStarGODA-MDP local, utilizando o *endpoint* "/formula/cost?id=GoalModel_1618949483234&goal=G1_T1_2", e seu retorno, contendo o nó "G1_T1_2" como raiz e seus respectivos nós filhos. A requisição foi feita utilizando o aplicativo Insomnia[17], passando o identificador da árvore da Listagem 3.3 e consultando custo.

Para implementar a requisição GET foram adicionadas duas novas rotas no *controller* de integração, uma para confiabilidade e outra para custo. As rotas chamam a classe *FormulaService*, que contém os métodos responsáveis por manusear a fórmula paramétrica. Primeiramente, a árvore é carregada do disco utilizando o identificador e a diferenciação entre custo e confiabilidade contida da rota. Adiante, é executado o algoritmo de busca em largura, que busca e retorna o nó requisitado, utilizando o parâmetro *goal* passado pelo usuário durante a requisição. Após a conclusão do algoritmo, é retornado o FormulaTreeNode correspondente. Caso contrário, é retornado um FormulaTreeNode vazio.

POST

A rota de POST implementa a criação de uma nova fórmula composicional e é disponibilizada sob o *endpoint* "/formula/<modo>", onde '<modo>' descreve se a árvore criada

será de custo ou confiabilidade. O corpo da requisição tem a estrutura apresentada na listagem 3.4, com "id" sendo o nome que será concatenado a um *timestamp* no momento da criação do arquivo, e o campo "tree" representando a árvore que será criada. Os métodos de *get* e *set* foram implementados como padrão na classe.

```
1 public class FormulaTreeModel {
2     private String id;
3     private FormulaTreeNode tree;
4 }
```

Listing 3.4: Classe *FormulaTreeModel*

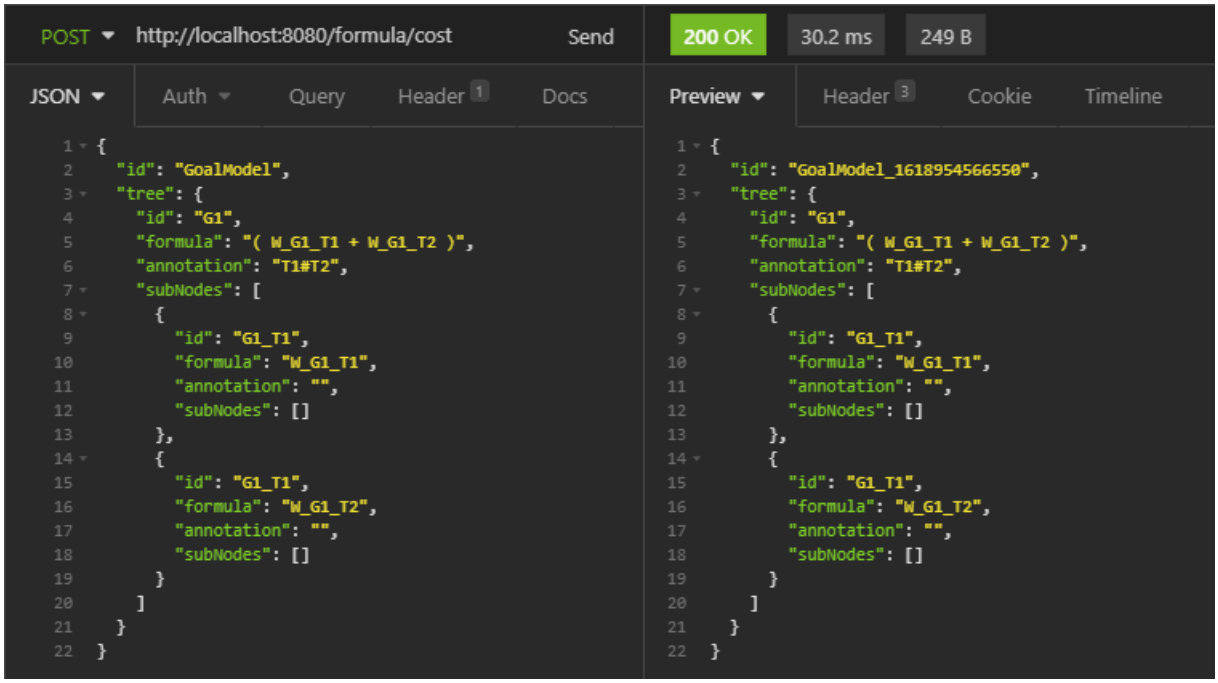

Figura 3.8: Requisição POST e retorno.

A Figura 3.8 exemplifica uma requisição POST para o PiStarGODA-MDP utilizando o aplicativo Insomnia, com seu retorno à direita da Figura. No corpo da requisição, à esquerda, é passado o identificador da árvore em 'id' e a árvore completa que deverá ser criada em 'tree'. O retorno da requisição contém: (i) o 'id' criado pelo servidor, que concatena o 'id' passado na requisição com um *timestamp* gerado, e (ii) a árvore que foi criada, caso a criação tenha sucedido. Caso ocorra algum erro durante a criação, os campos de retorno retornarão vazios.

Em sua implementação, a rota de POST recebe diretamente um *FormulaTreeModel*, o qual o *framework* Spring preenche de acordo com o JSON que foi enviado no corpo da requisição. Dessa forma, utilizando o parâmetro '<modo>' da rota, o servidor cria um

arquivo com nome no modelo '<id>_<timestamp>_<modo>.json' e escreve o corpo da árvore no mesmo. Se a criação ocorrer com sucesso, o servidor preenche um FormulaTreeNode com o 'id' gerado e com a árvore salva, retornando esse objeto como resposta da requisição. Se não obtiver sucesso, o retorno será um objeto vazio.

PUT

A rota de PUT implementa a edição de uma fórmula paramétrica composicional existente, e é disponibilizada no *endpoint* "/formula/<modo>?id=<identificador>&goal=<nó da árvore>&shouldPersist=<true|false>", onde <modo> pode ser "reliability" para confiabilidade ou "cost" para custo. Por sua vez, <identificador> é o identificador da árvore, gerado e retornado pelo PiStarGODA-MDP no momento de sua criação. Já <nó da árvore> é o identificador do nó da árvore onde se deseja fazer a modificação. Por fim, o parâmetro booleano "shouldPersist" exprime se a modificação deve ser persistida ou não. O retorno da requisição é a árvore inteira, com a fórmula atualizada até a raiz.

A rota de PUT permite uma variedade de edições sobre a árvore. As opções são: (i) mudar a *runtime annotation*, (ii) mudar a decomposição AND-OR, (iii) adicionar ou retirar o contexto e (iv) retirar ou adicionar uma tarefa. Cada uma dessas opções estão exemplificadas nas Figuras 3.9 e 3.10, 3.11 e 3.12, 3.13 e 3.14, 3.15 e 3.16, respectivamente. Os exemplos utilizam como base a árvore listada na Listagem 3.3 e o aplicativo Insomnia para fazer as requisições para a rota "/formula/reliability?id=GoalModel_1620691716217&goal=G1_T1_2&shouldPersist=false".

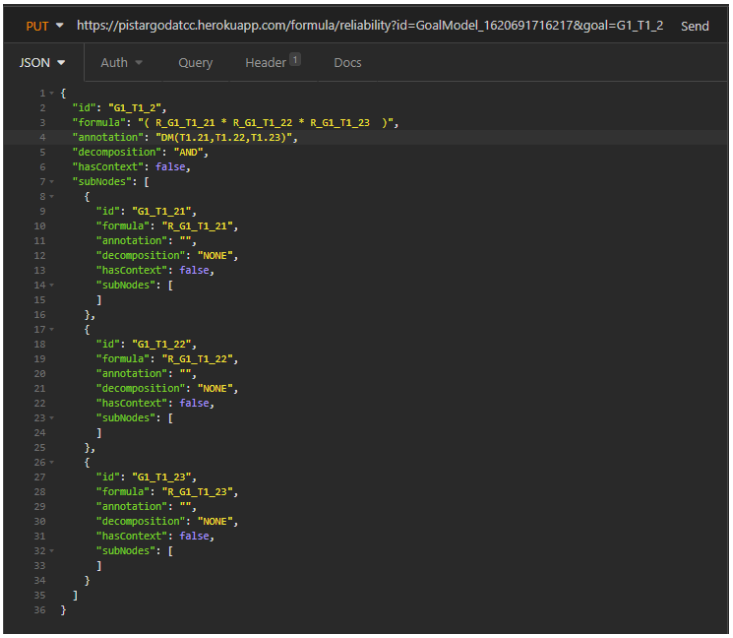

Figura 3.9: Requisição PUT para editar a *runtime annotation* do nó G1_T1_2.

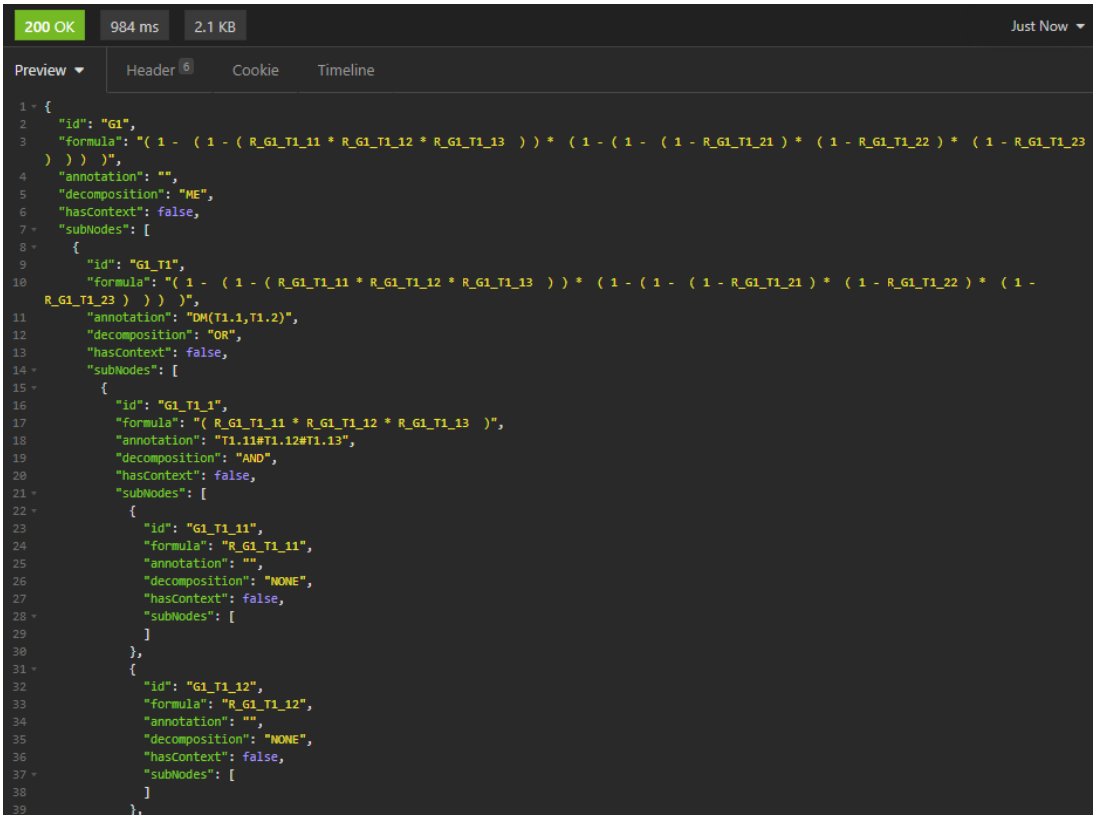

Figura 3.10: Fragmento do retorno da requisição PUT ao editar a *runtime annotation* do nó G1_T1_2.

As Figuras 3.9 e 3.10 contém um exemplo de uma requisição feita para o PiStarGODA-MDP com o intuito de editar a *runtime annotation* do nó "G1_T1_2", mudando-a de "T1.21#T1.22#T1.23", que, junto com a decomposição, representa um AND paralelo, para "DM(T1.21,T1.22,T1.23)" (*decision making*). A mudança faz com que a fórmula do nó mude, e essa mudança é propagada árvore acima, atualizando toda a árvore JSON da fórmula paramétrica.

As Figuras 3.11 e 3.12 contém o corpo da requisição para editar a decomposição AND do nó "G1_T1_2" para OR e seu retorno, causando uma mudança na fórmula. A mudança é aplicada no nó e posteriormente é aplicada no resto da árvore. Para editar a decomposição, não é necessário modificar a *runtime annotation*, uma vez que a mesma representa se a decomposição é paralela ou sequencial.

Por sua vez, as Figuras 3.13 e 3.14 exemplificam a edição da propriedade "hasContext" no corpo da requisição. Dessa forma, a fórmula presente no pai das tarefas passa a conter o contexto dessas tarefas, e, assim, a fórmula é atualizada em toda a árvore levando em consideração o contexto dessas tarefas, ou seja, adicionando a anotação de contexto das tarefas modificadas.

PUT https://pistargodatcc.herokuapp.com/formula/reliability?id=GoalModel_1620691716217&goal=G1_T1_2 Send

JSON Auth Query Header 1 Docs


```
{
  "id": "G1_T1_2",
  "formula": "( R_G1_T1_21 * R_G1_T1_22 * R_G1_T1_23  )",
  "annotation": "T1.21#T1.22#T1.23",
  "decomposition": "OR",
  "hasContext": false,
  "subNodes": [
    {
      "id": "G1_T1_21",
      "formula": "R_G1_T1_21",
      "annotation": "",
      "decomposition": "NONE",
      "hasContext": false,
      "subNodes": [
      ]
    },
    {
      "id": "G1_T1_22",
      "formula": "R_G1_T1_22",
      "annotation": "",
      "decomposition": "NONE",
      "hasContext": false,
      "subNodes": [
      ]
    },
    {
      "id": "G1_T1_23",
      "formula": "R_G1_T1_23",
      "annotation": "",
      "decomposition": "NONE",
      "hasContext": false,
      "subNodes": [
      ]
    }
  ]
}
```


Figura 3.11: Requisição PUT para editar a decomposição do nó G1_T1_2.

200 OK 218 ms 2.1 KB Just Now

Preview Header 6 Cookie Timeline


```
{
  "id": "G1",
  "formula": "( 1 -  ( 1 - ( R_G1_T1_11 * R_G1_T1_12 * R_G1_T1_13  ) ) *  ( 1 - ( 1 -  ( 1 - R_G1_T1_21 ) *  ( 1 - R_G1_T1_22 ) *  ( 1 - R_G1_T1_23 ) ) ) )",
  "annotation": "",
  "decomposition": "ME",
  "hasContext": false,
  "subNodes": [
    {
      "id": "G1_T1",
      "formula": "( 1 -  ( 1 - ( R_G1_T1_11 * R_G1_T1_12 * R_G1_T1_13  ) ) *  ( 1 - ( 1 -  ( 1 - R_G1_T1_21 ) *  ( 1 - R_G1_T1_22 ) *  ( 1 - R_G1_T1_23 ) ) ) )",
      "annotation": "DM(T1.1,T1.2)",
      "decomposition": "OR",
      "hasContext": false,
      "subNodes": [
        {
          "id": "G1_T1_1",
          "formula": "( R_G1_T1_11 * R_G1_T1_12 * R_G1_T1_13  )",
          "annotation": "T1.11#T1.12#T1.13",
          "decomposition": "AND",
          "hasContext": false,
          "subNodes": [
            {
              "id": "G1_T1_11",
              "formula": "R_G1_T1_11",
              "annotation": "",
              "decomposition": "NONE",
              "hasContext": false,
              "subNodes": [
              ]
            },
            {
              "id": "G1_T1_12",
              "formula": "R_G1_T1_12",
              "annotation": "",
              "decomposition": "NONE",
              "hasContext": false,
              "subNodes": [
              ]
```


Figura 3.12: Fragmento do retorno da requisição PUT ao editar a decomposição do nó G1_T1_2.

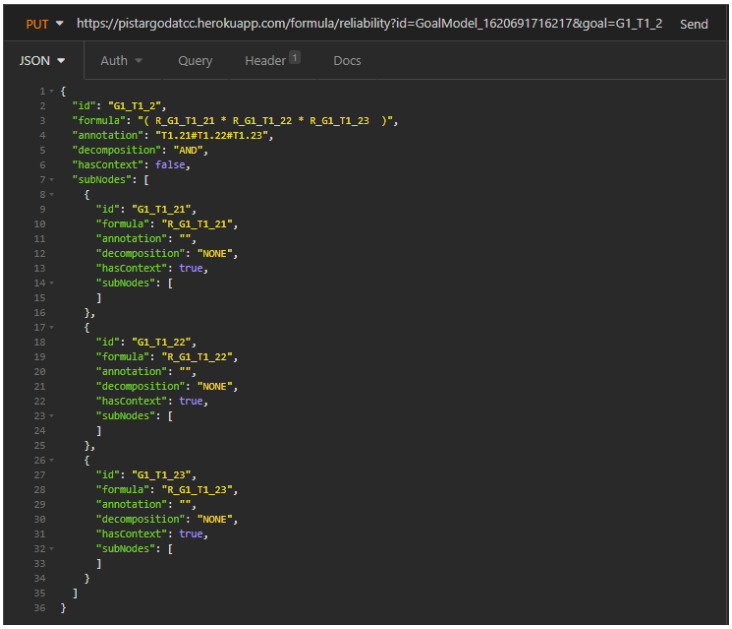

Figura 3.13: Requisição PUT para editar os contextos das *leaf tasks* do nó G1_T1_2.

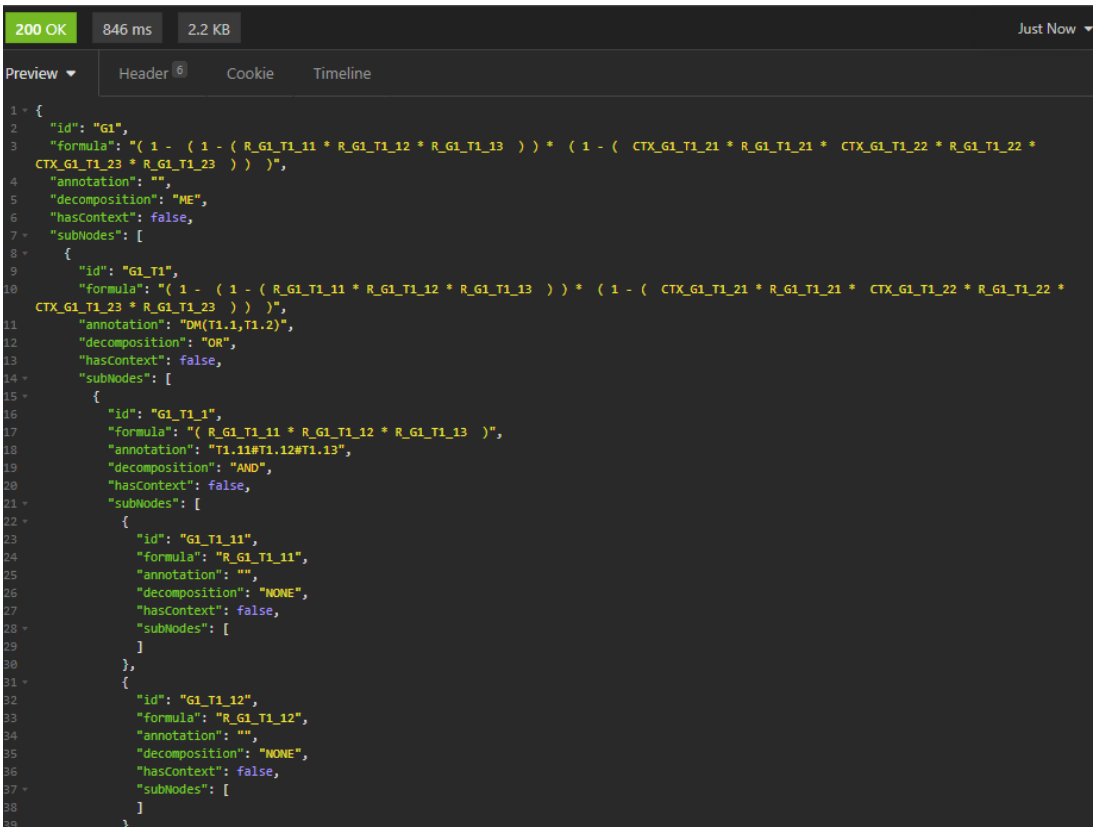

Figura 3.14: Fragmento do retorno da requisição PUT ao editar os contextos das *leaf tasks* do nó G1_T1_2.

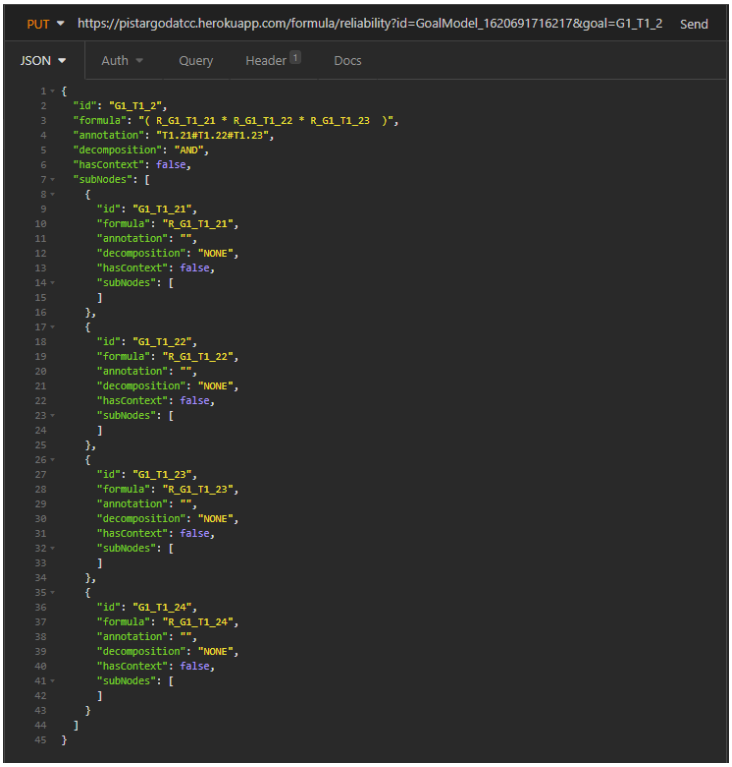

Figura 3.15: Requisição PUT para adicionar uma *leaf task* ao nó G1_T1_2.

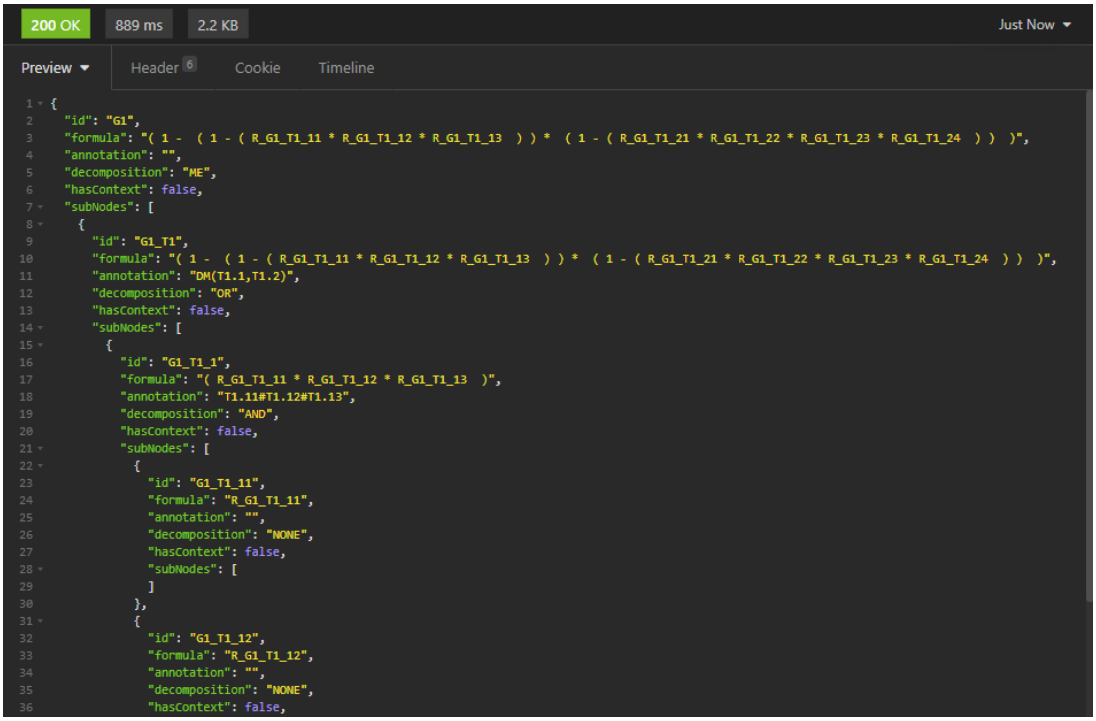

Figura 3.16: Fragmento do retorno da requisição PUT ao adicionar uma *leaf task* ao nó G1_T1_2.

Por fim, as Figuras 3.15 e 3.16 exemplificam uma adição de uma *leaf task* ao nó "G1_T1_2", não sendo necessário mudar a *runtime annotation*, uma vez que a mesma representa uma execução paralela. Em caso de adições em nós com execuções sequenciais, deve-se verificar a necessidade de editar também a *runtime annotation* do nó. É também possível remover uma tarefa ou objetivo, sendo que o processo é o mesmo: a mudança deve ser feita a partir do pai da tarefa/objetivo para que o servidor consiga propagar a mudança pela árvore consistentemente.

A implementação da rota PUT consiste em duas rotas, uma para confiabilidade e outra para custo, que recebem o nó que deve ser editado com uma nova versão. Então, o método responsável pela edição recupera a árvore do arquivo JSON salvo, utiliza um algoritmo de busca em largura para acessar o nó que deve ser editado e muda os valores para os valores passados no corpo da requisição. Por fim, esse método chama o método responsável por gerar a fórmula paramétrica do modelo de objetivos, passando as informações de contexto, *runtime annotation* e filhos diretos do nó. Dessa forma, o nó editado tem sua fórmula paramétrica atualizada corretamente.

Para propagar a mudança pela árvore, é utilizado um algoritmo de busca em profundidade a partir da raiz da árvore, substituindo o fragmento antigo de fórmula pelo novo, aproveitando a composicionalidade da fórmula. Portanto, não é necessário gerar toda a fórmula paramétrica novamente. Para tanto, utiliza-se um mecanismo muito parecido com aquele utilizado para gerar a fórmula inicialmente, implementada originalmente no PiStarGODA-MDP. A principal diferença entre os dois mecanismos é que o mecanismo utilizado pelo PUT substitui a fórmula por completo e não símbolos.

Pelo fato da fórmula ser calculada apenas uma vez para o nó e depois propagada por toda a árvore, a edição tem uma complexidade O(N*M) no pior caso, onde N é o número de nós e M é o tamanho das strings que terão que ser substituídas. Porém, o tamanho das strings diminui conforme a profundidade da árvore aumenta, portanto, em vista de se gerar uma fórmula novamente, essa abordagem possui um ganho no tempo, mesmo que a complexidade seja a mesma.

3.3.3 Fluxo de geração da fórmula paramétrica

Para gerar a fórmula paramétrica, utilizar a árvore e as rotas implementadas, é necessário acessar o PiStarGODA-MDP. A Figura 3.17 contém um exemplo com o modelo de objetivos construído utilizando a ferramenta. Não foi necessária nenhuma modificação no *frontend* do *framework*.

Após a criação, o usuário deve utilizar o botão "EPMC Formula Generation", apontado na Figura 3.18. Assim, o *frontend* faz uma requisição ao *backend* passando o modelo de objetivos para que o servidor gere as fórmulas paramétricas para confiabilidade e custo.

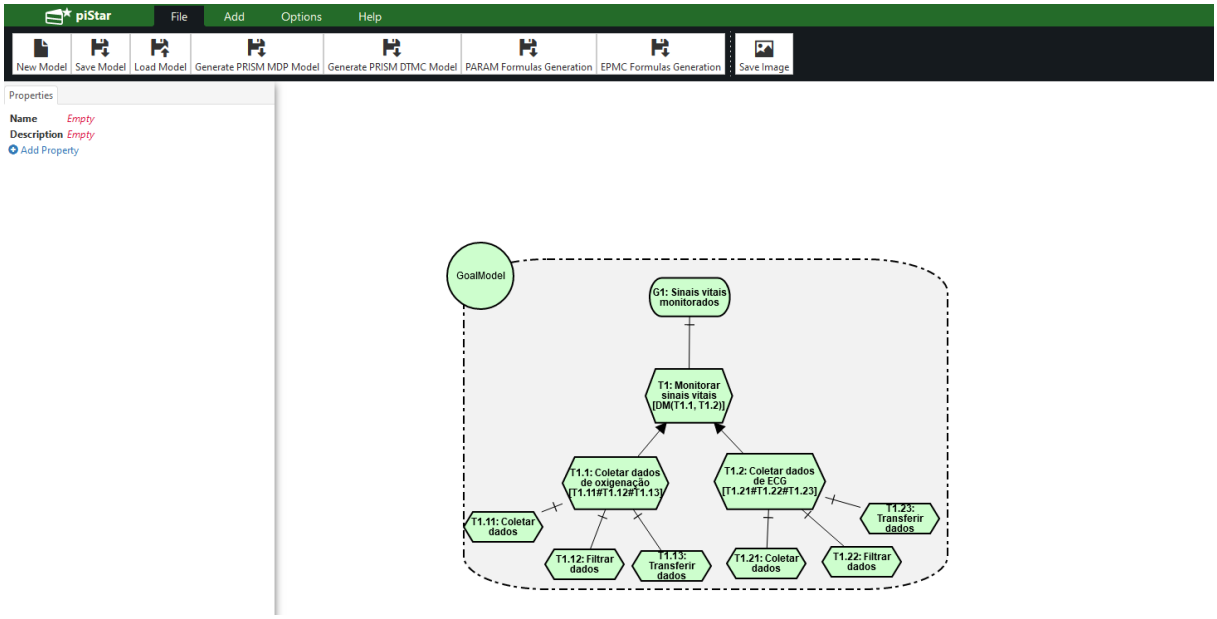

Figura 3.17: Criação de um modelo de objetivos no PiStarGODA-MDP.

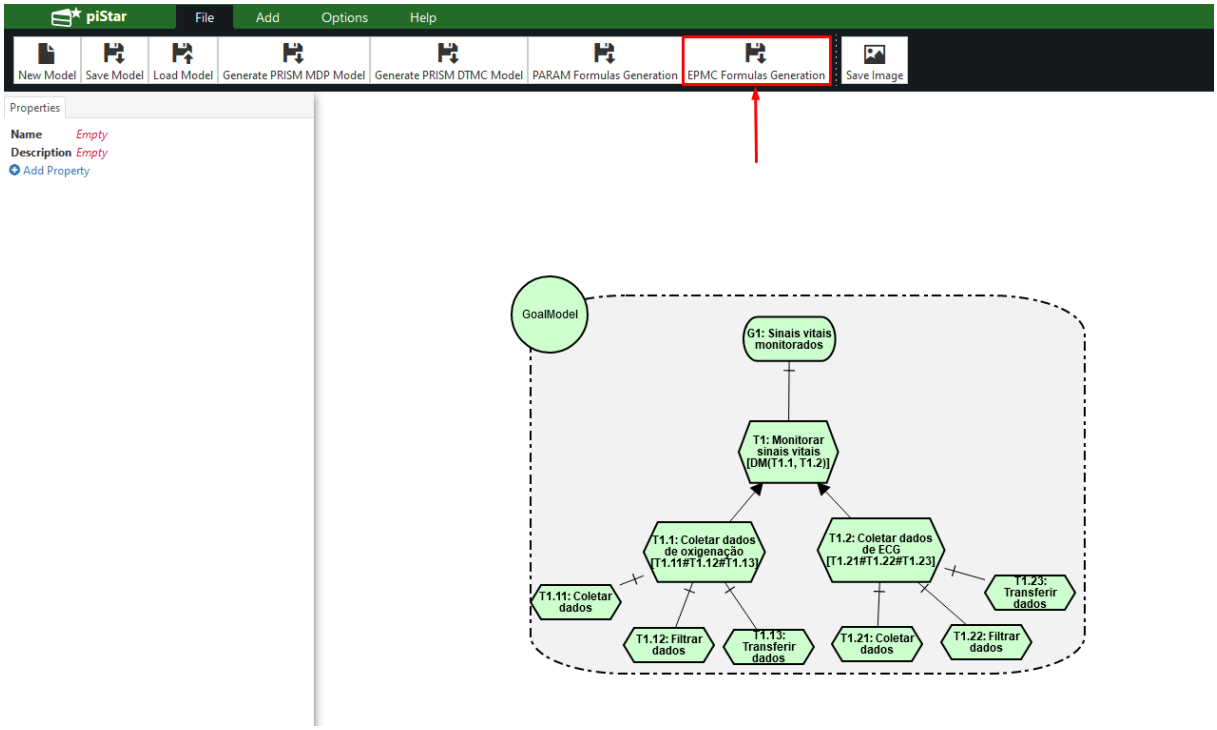

Figura 3.18: Botão utilizado para gerar as fórmulas paramétricas de custo e confiabilidade.

Em seguida, é disponibilizado ao usuário uma pasta compactada, que contém os arquivos gerados pelo PiStarGODA-MDP. Essa pasta contém as fórmulas paramétricas em árvore para custo e confiabilidade, chamados cost_tree.json e reliability_tree.json, e um

arquivo chamado formulas_id.out, que contém o identificador das árvores, utilizado nas requisições HTTP para o servidor. Dessa forma, o usuário pode utilizar tanto os arquivos contendo as árvores quanto o PiStarGODA-MDP para manuseio das fórmulas.

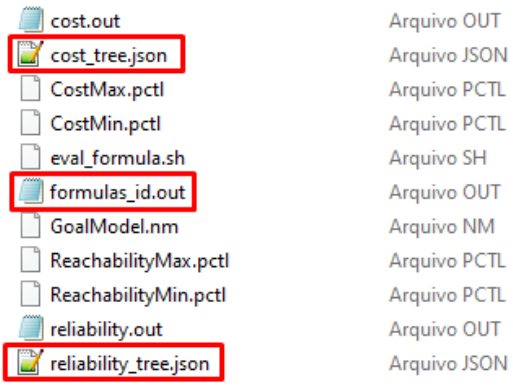

Figura 3.19: Pasta retornada pelo PiStarGODA-MDP após geração das fórmulas paramétricas.

Capítulo 4

Prova de Conceito

A prova de conceito visa mostrar a utilidade de se ter uma fórmula paramétrica composicional e a utilização da mesma como um serviço por meio do PiStarGODA-MDP. Este capítulo apresenta a prova de conceito feita no sistema da *Body Sensor Network* (BSN), utilizando a composicionalidade da fórmula para manuseá-la em tempo de execução, permitindo que o sistema não necessite regerar a fórmula a cada modificação.

Primeiramente, o capítulo apresenta a BSN e os módulos que a compõe, além de descrever o cenário do experimento e a motivação para se utilizar a fórmula composicional. Por fim, expõe os resultados após a implementação da solução e o uso do PiStarGODA-MDP com as novas rotas implementadas.

4.1 *Body Sensor Network*

A *Body Sensor Network* é um protótipo *open-source* de um sistema auto-adaptativo para o domínio de saúde [4], disponível no Github[1]. Construída em cima do *middleware* ROS [18] utilizando C++, a BSN implementa uma simulação de uma rede de sensores corporais [19] que coleta os dados de um paciente em um ambiente de UTI, e os envia para um *hub* central, chamado no sistema de *Centralhub*, responsável por fazer a fusão dos dados e detectar se o paciente está em uma situação emergencial. A BSN implementa o comportamento de seis sensores presentes em um ambiente de UTI: termômetro, oxímetro, eletrocardiógrafo, sensor de pressão diastólica, sensor de pressão sistólica e glicosímetro [4].

A BSN possui uma arquitetura composta pelos módulos *Managing System*, *Knowledge Repository*, *Managed System* e *Simulation*, como representado na Figura 4.1. Cada módulo será detalhado abaixo [4]:

[1] https://github.com/lesunb/bsn

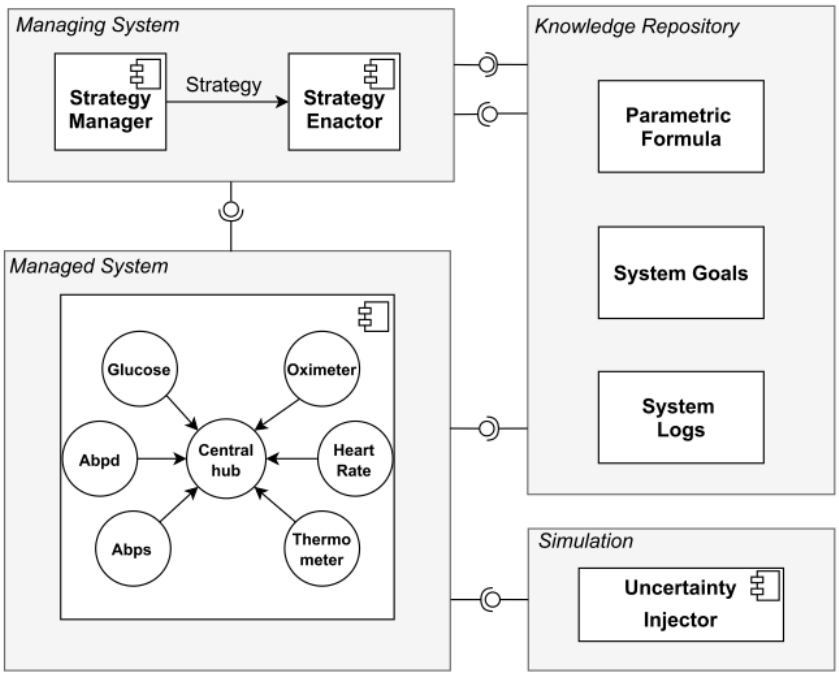

Figura 4.1: Arquitetura da BSN [4].

- ***Managing System***: Este módulo contém o *Strategy Enactor* e o *Strategy Manager* e é responsável por implementar o controlador que adapta o sistema conforme necessário. O *Strategy Manager* é responsável por estimar a confiabilidade e custo locais do sistema para se alcançar uma confiabilidade e custo globais, utilizando a fórmula paramétrica presente no *Knowledge Repository*. Por sua vez, o *Strategy Enactor* implementa o controlador de fato, e é responsável por aplicar estratégias de adaptação visando alcançar os valores de confiabilidade e custo locais definidos pelo *Strategy Manager*. Os componentes locais que compõem a confiabilidade geral do sistema são os sensores e o *Centralhub*.
- ***Managed System***: Contém os sensores, que monitoram os sinais vitais e o *Centralhub*. Cada sensor é responsável por escutar em um tópico ROS e coletar o dado, passar o mesmo por um filtro, verificar se o dado informa um estado de baixo, médio ou alto risco, e enviar o mesmo ao *Centralhub*. O *Centralhub*, por sua vez, faz a fusão dos dados recebidos de todos os sensores e verifica se o paciente está ou não em emergência. O *Managed System* contém também um nó ROS que é o responsável por gerar todos os dados coletados pelo sensor, chamado de nó paciente. O paciente gera os dados usando cadeias de Markov, com os estados representando faixas de risco para os valores gerados.

- ***Knowledge Repository***: O *Knowledge Repository* contém as fórmulas paramétricas para confiabilidade e custo, baseado em consumo de energia de cada sensor. Contém também os objetivos a serem atingidos e um sistema de *logging*, que persiste as informações sobre a execução geral do sistema, como os comandos de adaptação utilizados pelo *Enactor*, o status de cada sensor, os eventos de ativação e desativação dos sensores, informação sobre o uso de energia e as incertezas injetadas pelo *Uncertainty Injector*.
- ***Simulation***: Este módulo contém o *Uncertainty Injector* e é responsável por simular as incertezas no sistema, causando variação na confiabilidade do sistema. A injeção de incertezas é feita na etapa de coleta de dados dos utilizando mensagens ROS.

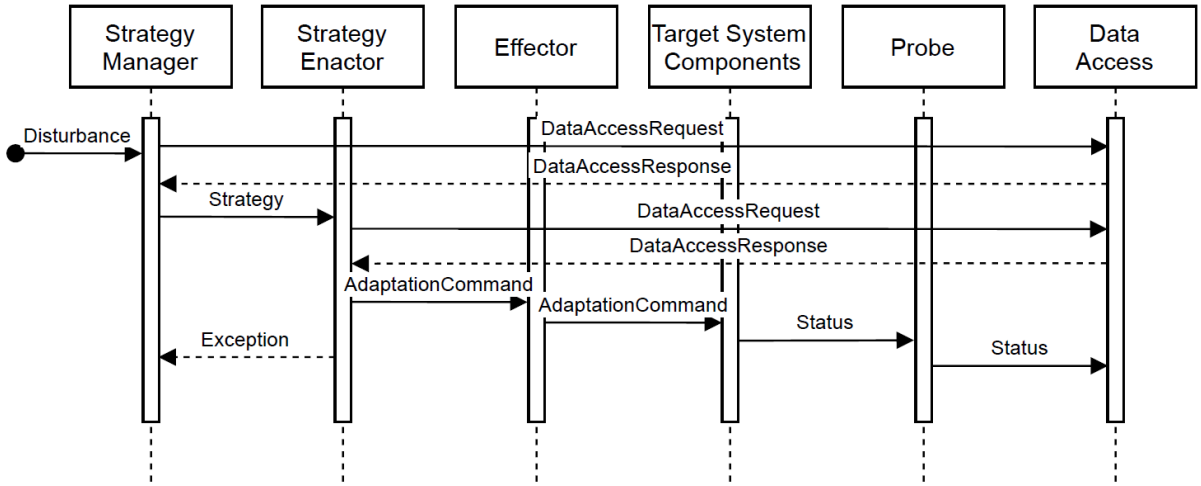

Figura 4.2: Diagrama de sequência de adaptação da BSN [4].

A Figura 4.2 contém o diagrama de sequência de adaptação da BSN que ilustra o processo de troca de mensagens quando uma adaptação é necessária. Uma adaptação se torna necessária quando há uma discrepância entre o valor atual e o desejado de um atributo de interesse. Se a adaptação é necessária, o *Strategy Manager* envia uma requisição para o componente *DataAccess*, responsável por buscar nos *logs* de execução a informação sobre a taxa de erros dos componentes do *Managed System* ou do consumo de bateria de cada um. Então, o *Strategy Manager* estima os valores desejados para cada componente e os envia para o *Enactor*. O *Enactor* utiliza estes valores para estimar a frequência de cada componente e envia um *Adaptation Command* para o *Effector*, que é responsável por redirecionar o comando de adaptação para cada componente. Por fim, os componentes do *Managed System* recebem os comandos de adaptação, mudam suas

frequências de acordo e enviam seus status para o *Probe*, que redireciona os dados para o *DataAccess*, responsável por logar os mesmos [4].

```
<launch>
    <!-- Blood Oxigenation Measurement Sensor -->
    <node name="patient" pkg="patient" type="patient" output="screen" />

    <param name="frequency" value="10" />

    <param name="vitalSigns" value="oxigenation, heart_rate, temperature, abps, abpd, glucose" />

    <!-- Frequency for changes in states of each markov in Hertz -->
    <param name="oxigenation_Change" value="0.2"/>
    <param name="heart_rate_Change" value="0.1"/>
    <param name="temperature_Change" value="0.1"/>
    <param name="abps_Change" value="0.1"/>
    <param name="abpd_Change" value="0.1"/>
    <param name="glucose_Change" value="0.1"/>

    <!-- Offsets for each changes, in seconds -->
    <param name="oxigenation_Offset" value="10"/>
    <param name="heart_rate_Offset" value="10"/>
    <param name="temperature_Offset" value="10"/>
    <param name="abps_Offset" value="10"/>
    <param name="abpd_Offset" value="10"/>
    <param name="glucose_Offset" value="10"/>

    <!-- Markov chain for oxigenation -->
    <param name="oxigenation_State0" value="0,0,0,0,0" />
    <param name="oxigenation_State1" value="0,0,0,0,0" />
    <param name="oxigenation_State2" value="0,0,90,8,2" />
    <param name="oxigenation_State3" value="0,0,75,10,5" />
    <param name="oxigenation_State4" value="0,0,5,35,60" />

    <!-- Risk values for oximeter -->
    <param name="oxigenation_HighRisk0" value="-1,-1" />
    <param name="oxigenation_MidRisk0" value="-1,-1" />
    <param name="oxigenation_LowRisk" value="65,100" />
    <param name="oxigenation_MidRisk1" value="55,65" />
    <param name="oxigenation_HighRisk1" value="0,55" />
```

Figura 4.3: Fragmento do arquivo de configuração do paciente [4].

Para realizar configuração dos módulos, o ROS provê um arquivo de configuração em formato xml, exemplificado pela Figura 4.3. O arquivo de configuração contém informações sobre o nó ROS e parâmetros que serão carregados pelo ROS durante a inicialização dos nós. Os nós conseguem acessar as configurações presentes no arquivo por meio do ROS a qualquer momento. Na BSN isso é feito no momento de inicialização, sendo o *setUp* o primeiro método a ser chamado.

4.2 Cenário

A BSN possui seis sensores utilizados para monitorar os sinais vitais do paciente. Tais sensores estão representados no modelo de objetivos da BSN como *leaf tasks*, apresentado na Figura 4.4, com T1.1 para o oxímetro, T1.2 para o ECG, T1.3 para o termômetro, T1.4 para o sensor de pressão sistólica, T1.5 para o sensor de pressão diastólica, e T1.6 para o sensor de glicose. Todas essas *leaf tasks* são decomposições AND da *task* de monitorar sinais vitais e, portanto, um sensor que é desligado ocasiona uma confiabilidade geral em zero, mesmo que seja do interesse do usuário desligar este sensor. A fórmula paramétrica

abaixo foi gerada pelo PiStarGODA-MDP a partir do modelo de objetivos da BSN e assim, é possível notar que cada componente tem um peso muito alto no cálculo da fórmula.

$$((CTX\_G3\_T1\_1 * R\_G3\_T1\_1 * CTX\_G3\_T1\_2 * R\_G3\_T1\_2 * CTX\_G3\_T1\_3 * R\_G3\_T1\_3 * CTX\_G3\_T1\_4 * R\_G3\_T1\_4 * CTX\_G3\_T1\_5 * R\_G3\_T1\_5 * CTX\_G3\_T1\_6 * R\_G3\_T1\_6) * CTX\_G4\_T1 * R\_G4\_T1) \quad (4.1)$$

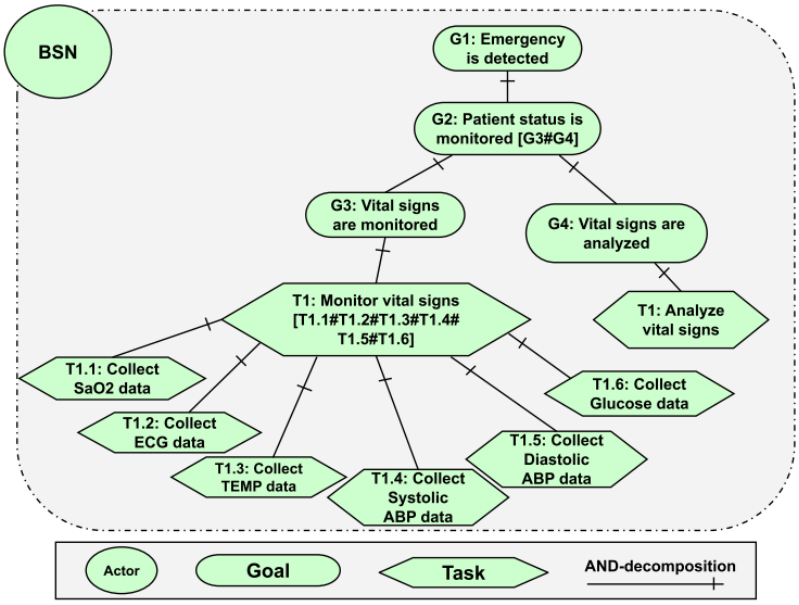

Figura 4.4: Modelo de objetivos da BSN [4].

Partindo da limitação do modelo de objetivos da BSN e o fato de um sensor que é desativado propositalmente fazer com que a confiabilidade geral do sistema vá para zero, esta prova de conceito visa mostrar que utilizar uma fórmula paramétrica composicional em formato de árvore traz benefícios, como permitir mudanças na fórmula em tempo de execução mais facilmente. Dessa forma, este experimento, que simula um cenário de manutenção para desligar um sensor, usará a propriedade composicional da fórmula e as rotas implementadas no PiStarGODA-MDP para modificar a fórmula paramétrica para que a mesma não leve em consideração o sensor que será desligado e não haja impacto na confiabilidade do sistema, permitindo que sensores sejam desativados pelo usuário sem a necessidade de um desenho de um novo modelo de objetivos.

4.3 Experimento

O experimento, utilizado como prova de conceito, simulará um cenário onde o sensor de temperatura, G3_T1_3, está com a confiabilidade muito baixa, e então, o usuário manualmente o desliga. Tal cenário simula um caso real, onde sensores são desligados e trocados com o intuito de manutenção no sistema. O desligamento do sensor ocorrerá sem que a BSN seja reiniciada, e, portanto, não será possível gerar um novo modelo de objetivos, sendo necessária a utilização da rota de PUT implementada no PiStarGODA-MDP para corrigir a fórmula utilizada no cálculo da confiabilidade da BSN, para que a mesma não contenha o sensor desligado.

Para realizar o experimento, foi implementado um novo nó ROS no módulo *Knowledge Repository*, denominado *Formula Accessor*, responsável por encapsular o cliente responsável por fazer as requisições com o PiStarGODA. O *Formula Accessor* contém três *subscribers*, que se inscrevem nos tópicos "/get_formula", "/add_task" e "/remove_task", e que são responsáveis por receberem os parâmetros para retornar a fórmula atualizada, adicionar uma nova *task* e remover uma *task*. Como parâmetro, os tópicos recebem o objetivo ou tarefa do modelo de objetivos que deve ser editado ou consultado. O *Formula Accessor* publica no tópico "/formula" toda vez que recebe uma requisição de consulta ou uma requisição de edição. Dessa forma, a mudança na fórmula acontece de forma assíncrona, por meio de *publishers* e *subscribers*, mitigando a latência causada por uma requisição HTTP.

Ao se desligar um sensor, o *Data Access* recebe desse sensor uma mensagem avisando seu desligamento e persiste o evento de desativação em um arquivo. Periodicamente, o *System Manager* consulta o *Data Access* para calcular a confiabilidade geral e para verificar os sensores ativos. Ao detectar que um sensor foi desligado, o *System Manager* envia uma mensagem ao *Formula Accessor* solicitando que a fórmula seja atualizada, retirando o sensor que foi desligado. Para tal, o *Formula Accessor* faz uma requisição PUT ao PiStarGODA-MDP, recebendo como resposta a árvore da fórmula atualizada. Após o retorno, o *Formula Accessor* atualiza sua fórmula cacheada e publica no nó "/formula" a fórmula atualizada. O *Data Access*, que é inscrito nesse tópico, recebe a fórmula atualizada e a disponibiliza ao *System Manager*, que passa a utilizar a fórmula correta.

Para fazer a simulação, o nó *Injector* foi configurado de forma a injetar mais erros no sensor G3_T1_3, de forma a simular constantes falhas no sensor. Assim, o G3_T1_3 falha incessantemente, simulando um defeito, e justificando o desligamento do mesmo por parte do usuário, por ser um ponto crítico, assim como os outros sensores. O *Injector* foi configurado de modo a injetar incertezas após um minuto de execução, em um número excessivo, como demonstrado na Figura 4.5.

```
<!--Parameters for g3t1_3 uncertainty injection-->
<param name="g3t1_3/type" value="random" />                    <!--Could be: step, ramp or random-->
<param name="g3t1_3/offset" value="1" />
<param name="g3t1_3/amplitude" value="1" />
<param name="g3t1_3/frequency" value="1" />                 <!-- 0.25 Hz -->
<param name="g3t1_3/duration" value="3" type="int" />          <!-- not important for random input -->
<param name="g3t1_3/begin" value="60" type="int" />             <!-- (s) instant to begin the injection-->
```

Figura 4.5: Configuração do *Injector* para o G3_T1_3.

No experimento, a BSN é iniciada com todos os sensores funcionando e com uma injeção moderada de incertezas em cada. Ao se passar um minuto, o *Injector* passa a injetar um elevado número de incertezas no sensor termômetro, G3_T1_3, fazendo com que a confiabilidade do mesmo vá para zero, o que faz com que a confiabilidade da BSN como um todo também vá para zero. Assim, a BSN tenta achar algum plano de adaptação mas não obtém sucesso, uma vez que o sensor está com muitas falhas. Dessa forma, o sensor é desligado para que a confiabilidade geral do sistema se estabilize.

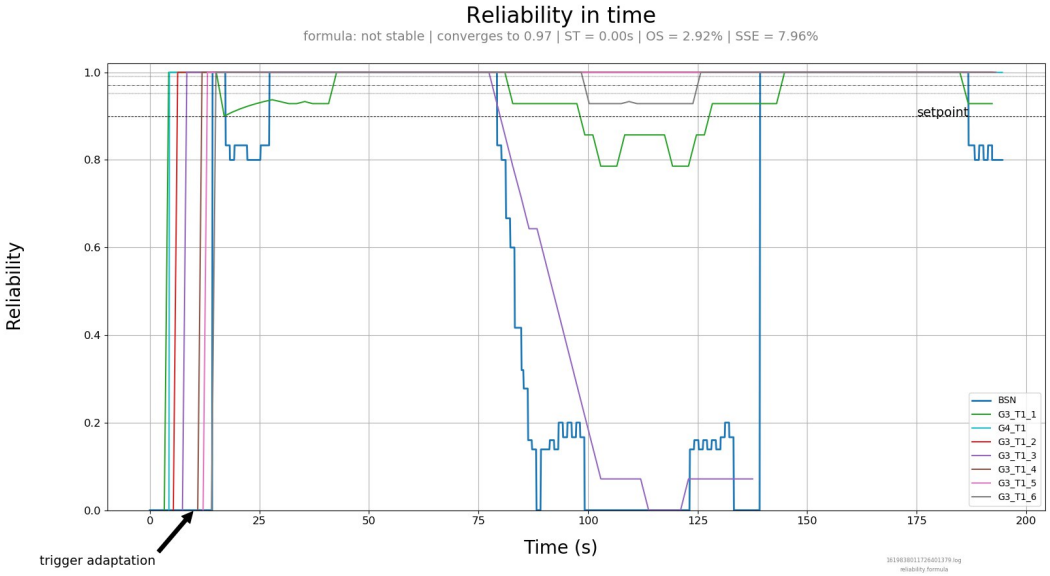

Figura 4.6: Gráfico de confiabilidade gerado após o experimento.

A Figura 4.6 apresenta o resultado gráfico do experimento, contendo um gráfico com as confiabilidades de cada componente e da BSN como um todo, onde G3_T1_3 tem sua confiabilidade em um valor bem baixo, fazendo com que a confiabilidade do sistema diminua. Ao analisar o gráfico, é possível notar que a confiabilidade do sistema volta a um valor alto logo após o desligamento do sensor defeituoso, aos 135 segundos aproximadamente. Após um breve momento, aos 140 segundos aproximadamente, a confiabilidade do sistema volta a um patamar aceitável, indicando que a fórmula foi atualizada com

sucesso, excluindo o sensor defeituoso. Dessa forma, o experimento demonstra a utilidade das modificações implementadas no PiStarGODA-MDP, funcionando como uma prova de conceito.

Capítulo 5

Conclusão

O objetivo principal deste trabalho foi prover a fórmula paramétrica como um serviço no PiStarGODA-MDP, de forma a facilitar a utilização da mesma em tempo de execução. Desse modo, o primeiro passo foi compor a fórmula paramétrica em formato de árvore, com isomorfismo com o modelo de objetivos, e salvar essa estrutura em um arquivo JSON. Após isso, as rotas GET, POST e PUT foram adicionadas ao *IntegrationController*, implementando a consulta, criação e edição de uma fórmula em árvore, respectivamente.

Ao prover a fórmula paramétrica do PiStarGODA-MDP como um serviço, torna-se possível utilizar a propriedade de composicionalidade da mesma em tempo de execução. O cenário aplicado no estudo de caso demonstra que, de fato, a implementação no PiStarGODA-MDP possui utilidade por editar a fórmula em tempo de execução fazendo apenas uma requisição PUT ao servidor. Assim, o experimento serve como prova de conceito para a implementação no PiStarGODA-MDP e para a utilização da fórmula paramétrica em árvore.

Como limitação, o trabalho encontra dificuldades em ser disponibilizado como a antiga versão do PiStarGODA-MDP, disponível na plataforma em nuvem Heroku[1], pois a modificação realizada neste trabalho utiliza arquivos salvos em discos para persistir as árvores, e o Heroku realiza varreduras periódicas em arquivos para exclusão. Dessa forma, o ideal seria a utilização de um serviço de disco, que em sua grande maioria são pagos, como o Amazon S3[2].

Em suma, o trabalho estende com sucesso o PiStarGODA-MDP, fornecendo a fórmula paramétrica como serviço e a disponibilizando em formato de árvore. Assim, o PiStarGODA-MDP passa a ser uma ferramenta presente na modelagem e em tempo de execução do sistema modelado.

[1] https://www.heroku.com/about
[2] https://aws.amazon.com/pt/s3/

Referências

[1] Mendonça, Danilo Filgueira, Genaína Nunes Rodrigues, Raian Ali, Vander Alves e Luciano Baresi: *Goda: A goal-oriented requirements engineering framework for runtime dependability analysis*. Information and Software Technology, 80:245–264, 2016, ISSN 0950-5849. `https://www.sciencedirect.com/science/article/pii/S0950584916301471`. ix, 1, 4, 5, 7, 8, 9

[2] Bergmann, Leandro Santos: *pistar-goda: Integração entre os projetos pistar e goda*. março 2018. `https://bdm.unb.br/handle/10483/20428`. ix, xi, 1, 6, 7, 9

[3] Solano, Gabriela Félix: *A goal-oriented approach to support the assurance process of self-adaptive systems under uncertainty*. 2019. ix, 1, 6, 9, 10, 11

[4] Gil, Eric Bernd, Ricardo Caldas, Arthur Rodrigues, Gabriel Levi Gomes da Silva, Genaína Nunes Rodrigues e Patrizio Pelliccione: *Body sensor network: A self-adaptive system exemplar in the healthcare domain*, 2021. x, 32, 33, 34, 35, 36

[5] Pimentel, Joao e Jaelson Castro: *pistar tool–a pluggable online tool for goal modeling*. Em *2018 IEEE 26th International Requirements Engineering Conference (RE)*, páginas 498–499. IEEE, 2018. 1, 7

[6] Bresciani, Paolo, Anna Perini, Paolo Giorgini, Fausto Giunchiglia e John Mylopoulos: *Tropos: An agent-oriented software development methodology*. Autonomous Agents and Multi-Agent Systems, 8:203–236, maio 2004. 4

[7] Ali, Raian, Fabiano Dalpiaz e Paolo Giorgini: *A goal-based framework for contextual requirements modeling and analysis*. Requirements Engineering, 15, novembro 2010. 5

[8] Dalpiaz, Fabiano, Alexander Borgida, Jennifer Horkoff e John Mylopoulos: *Runtime goal models: Keynote*. páginas 1–11, maio 2013, ISBN 978-1-4673-2914-9. 5

[9] Solano, Gabriela Félix: *Verificando a boa formação de modelos goda*. janeiro 2017. `https://bdm.unb.br/handle/10483/15776`. 6

[10] Puterman, Martin L: *Markov decision processes*. Handbooks in operations research and management science, 2:331–434, 1990. 6

[11] Kwiatkowska, Marta, Gethin Norman e David Parker: *Prism: Probabilistic symbolic model checker*. Em *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, páginas 200–204. Springer, 2002. 6, 7, 11

[12] Daws, Conrado: *Symbolic and parametric model checking of discrete-time markov chains*. Em *International Colloquium on Theoretical Aspects of Computing*, páginas 280–294. Springer, 2004. 7

[13] Ciesinski, Frank e Marcus Größer: *On probabilistic computation tree logic*. Em *Validation of Stochastic Systems*, páginas 147–188. Springer, 2004. 7

[14] Calinescu, Radu, Colin Alexander Paterson e Kenneth Johnson: *Efficient parametric model checking using domain knowledge*. IEEE Transactions on Software Engineering, 2019. 7

[15] Hahn, Ernst, Holger Hermanns, Björn Wachter e Lijun Zhang: *Param: A model checker for parametric markov models*. páginas 660–664, julho 2010, ISBN 978-3-642-14294-9. 7, 10

[16] Johnson, Rod, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Alef Arendsen, Darren Davison, Dmitriy Kopylenko, Mark Pollack *et al.*: *The spring framework–reference documentation*. interface, 21:27, 2004. 8

[17] Inc., Kong: *Insomnia*. `https://insomnia.rest/`. 22

[18] Quigley, Morgan, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler e Andrew Y Ng: *Ros: an open-source robot operating system*. Em *ICRA workshop on open source software*, volume 3, página 5. Kobe, Japan, 2009. 32

[19] Lo, Benny PL, Surapa Thiemjarus, Rachel King e Guang Zhong Yang: *Body sensor network–a wireless sensor platform for pervasive healthcare monitoring*, 2005. 32