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

# Monitoramento de Propriedades em Tempo Real: Um Estudo de Caso na Body Sensor Network

Samuel Andrade do Couto

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

# Monitoramento de Propriedades em Tempo Real: Um Estudo de Caso na Body Sensor Network

Samuel Andrade do Couto

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

Prof.a Dr.a Genaína Nunes Rodrigues (Orientadora)
Universidade de Brasília

Prof. Dr. Vander Ramos Alves
Universidade de Brasília

Prof. Dr. Rodrigo Bonifácio de Almeida
Universidade de Brasília

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

Brasília, 27 de outubro de 2021

# Dedicatória

Dedico este trabalho a Deus, que em sua sabedoria me permitiu ingressar na universidade, à minha família, que me apoiou durante cada momento dessa jornada, ao Núcleo de Vida Cristã (NVC), que tornou minha estadia na universidade tão mais leve e memorável, e aos meus amigos de curso, que partilharam comigo as lutas e conquistas.

# Agradecimentos

Agradeço a Deus pela oportunidade de chegar neste ponto de minha caminhada e por me proporcionar tantas alegrias, como a conclusão de um trabalho como este.

Agradeço à professora Genaína Rodrigues, que durante toda o período de orientação foi sempre motivadora, mesmo em momentos em que parecia que o trabalho não seria concluído.

Agradeço aos meus colegas do Ladecic, especialmente ao Eric Gil e ao Ricardo Caldas pelo tempo gasto me ajudando com o desenvolvimento do meu trabalho.

Agradeço aos meus pais Kennedy e Adriana, que durante todo o momento me incentivaram a concluir este trabalho, mesmo em momentos que me encontrava desanimado com situações alheias.

Agradeço a todos os meus amigos e colegas que contribuíram, direta ou indiretamente, com a produção deste trabalho. Desde pequenas ideias trocadas, até conselhos mais profundos, cada um teve uma contribuição que me permitiu terminá-lo.

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

A implementação de sistemas em tempo real pode ser bastante complexa no que se refere à garantia de suas propriedades em tempo de execução. Para tanto, abordagens baseadas em verificação de modelos são bastante adequadas. Nestas abordagens, primeiramente modela-se formalmente o sistema alvo utilizando ferramentas de verificação de modelos (e.g. UPPAAL), e então verificam-se as propriedades de interesse antes da implementação propriamente dita do sistema. Observa-se no entanto, que tal abordagem pode abstrair informações do sistema implementado não considerando aspectos físicos relevantes a serem considerados. Com isso, é importante que haja uma forma adequada de permitir que propriedades especificadas em verificadores de modelo sejam mapeadas adequadamente ao ambiente de implementação do sistema alvo. Para tanto, torna-se necessário o uso do conceito de monitores que observam o sistema tendo como referência as propriedades previamente verificadas e que devem ser asseguradas em tempo de execução. Neste trabalho, exercitamos este processo tendo como referência princípios de compreensão de programas, onde implementamos no Robot Operating System (ROS) monitores que observam propriedades previamente especificadas em UPPAAL. Para este trabalho, utilizou-se a Body Sensor Network (BSN), um protótipo de um sistema em tempo real implementado no ROS cujo objetivo é monitorar estados de saúde de pacientes e detectar estados de emergência por meio de uma rede sensores sem fio. Para validar a implementação, foram realizados experimentos para se obter resposta às seguintes perguntas: 1) Como monitorar propriedades de sistema na BSN em tempo real no ROS preservando sua arquitetura? 2) Dada a natureza embarcada da BSN, qual o gargalo computacional para se monitorar essas propriedades? Conclui-se mostrando que é possível realizar tal monitoramento, porém sua viabilidade depende no contexto em qual o sistema estará implementado.

**Palavras-chave:** Monitoramento de sistemas em tempo real, compreensão de programa, sistemas auto-adaptativos

# Abstract

*Implementing real-time systems can be quite complex when it comes to guaranteeing their runtime properties. To this end, approaches based on model checking are quite suitable. In these approaches the target system is first formally modeled using model checking tools (e.g. UPPAAL), and then verify the properties of interest before the actual implementation of the system. It is noted however, that such an approach may abstract information from the implemented system by not considering relevant physical aspects to be considered. Thus, it is important to have a suitable way to allow properties specified in model checkers to be properly mapped to the target system implementation environment. To this end, it becomes necessary to use the concept of monitors that observe the system with reference to the properties previously checked and that must be ensured at runtime. In this work, we exercise this process with reference to principles of program comprehension, where we implement in the Robot Operating System (ROS) monitors that observe properties previously specified in UPPAAL. For this work, we used the Body Sensor Network (BSN), a prototype of a real-time system implemented in the ROS whose objective is to monitor patients' health states and detect emergency states through a wireless sensor network. To validate the implementation, experiments were conducted to obtain answers to the following questions: 1) How to monitor system properties in BSN in real time in ROS while preserving its architecture? 2) Given the embedded nature of the BSN, what is the computational bottleneck to monitor these properties? We conclude by showing that it is possible to perform such monitoring, but its feasibility depends on the context in which the system will be implemented.*

**Keywords:** Runtime monitoring, program comprehension, self-adaptive systems

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Capítulo 1

# Introdução

Ao longo dos anos, os sistemas de software têm se tornado cada vez maiores e mais complexos, ao ponto em que tanto a realização de seu projeto, quanto sua manutenção, têm se tornado tarefas cada vez mais difíceis. Por um lado, ao planejar um sistema, o engenheiro de software responsável por essa tarefa deve considerar uma série de cenários e configurações diferentes que o sistema poderá assumir durante sua execução, com o objetivo de tentar prevenir o máximo possível de possíveis cenários problemáticos que poderiam, eventualmente, resultar em falhas de sistema. Por outro lado, a manutenção desses sistemas consiste na verificação e possível alteração de múltiplos parâmetros, o que demanda um esforço contínuo de manter uma documentação consistente e atualizada.

Nesse contexto, sistemas auto-adaptativos (SAS) podem ser utilizados como uma solução para resolver uma parte desses problemas. Estes são sistemas que conseguem, em tempo de execução, ajustar seu próprio comportamento ou configurações em resposta àquilo que é observado sobre o ambiente em que se encontra, o sistema em si e seus objetivos [3]. Uma das premissas desse tipo de sistema é a de reduzir a quantidade de esforço feita pelos engenheiros durante o tempo de *design* (e.g., considerar múltiplas possibilidades que poderiam causar uma falha de sistema.

A abordagem auto-adaptativa é utilizada em sistemas em que a ação humana é mais difícil de ser executada, como em aplicações espaciais, ou até mesmo quando é interessante ao sistema manter controle sobre múltiplos recursos, como no caso de veículos aéreos não-tripulados (UAVs). Um uso interessante dos SAS é em sistemas críticos, que são sistemas cuja falha pode resultar em perda de vidas, danos significativos a propriedades, ou danos ao ambiente [4]. Aparelhos médicos, especialmente aqueles que lidam com o monitoramento de sinais vitais, podem ser considerados sistemas críticos, pelo fato de que erros de monitoramento podem causar, em certas instâncias, o óbito de um paciente. Estes aparel+hos, portanto, são candidatos interessantes para uma abordagem auto-adaptativa.

Um aspecto importante desse tipo de sistema é assegurar que algumas propriedades

deste estão sendo satisfeitas. Uma solução interessante em tempo de *design* de sistema é modelá-lo juntamente com propriedades desejáveis (e.g., garantir que o modelo de sistema proveja um nível de confiabilidade mínimo). Para isso, existem ferramentas já consagradas no meio acadêmico, como a ferramenta UPPAAL [5, 6], que permite a modelagem e verificação de sistemas tempo real, além de suas propriedades especificadas em lógica de árvore computacional (CTL) e autômatos temporais. Dessa forma, é possível validar qual abordagem é a mais apropriada para o desenho de um sistema. Outra forma de tentar analisar esse problema é utilizando verificação de propriedades desejadas durante o tempo de execução do sistema.

A *Body Sensor Network (BSN)* [7] é um artefato de um sistema auto-adaptativo desenvolvido pela equipe do LADECIC (grupo de pesquisa no CNPq), e vem sendo utilizado como artefato-base para muitos trabalhos científicos [8, 9, 10], de diversas áreas de pesquisa, que vão desde a engenharia de software à teoria de controle. Alguns desses trabalhos tiveram como passos intermediários ou finais a modelagem da arquitetura de software da BSN para verificar sua confiabilidade e outras métricas importantes para sistemas críticos. Um desses trabalhos[10] resultou, entre outras coisas, em um modelo UPPAAL desse sistema, como também na modelagem de propriedades desejadas para o mesmo.

Este trabalho tem o objetivo de explorar a implementação de monitores que observam tais propriedades em tempo de execução no contexto da BSN. Dada a complexidade da arquitetura da BSN no ROS, utilizou-se conceitos de noções de programa para permitir um entendimento adequado de como introduzir tais monitores a partir dos módulos já existentes na BSN.

Em seguida, foram realizados experimentos para avaliar tal solução, verificando se, de fato, os novos componentes criados estão sendo capazes de eficazmente monitorar as propriedades de interesse. Com isso, por meio de tais experimentos deseja-se responder as seguintes perguntas:

1. Como monitorar propriedades de sistema na BSN em tempo real no ROS preservando sua arquitetura?
2. Dada a natureza embarcada da BSN, qual o gargalo computacional para se monitorar essas propriedades?

Para responder à primeira pergunta, busca-se criar uma solução que seja viável do ponto de vista da implementação da BSN no ROS (*Robot Operating System* [11] (com suas especificidades de linguagem, arquitetura, etc), mantendo sua arquitetura original. Já para a segunda pergunta, utilizou-se a ferramenta Stacer para verificar o consumo de recursos computacionais.

Este trabalho é organizado da seguinte forma: o Capítulo 2 conta com o referencial teórico abordando os temas necessários para a compreensão do problema e do artefato utilizado para solucioná-lo; o Capítulo 3 mostra a metodologia utilizada para criar a solução; no Capítulo 4, são apresentados os experimentos que foram elaborados para responder as questões propostas neste capítulo, além de mostrar os seus resultados e a resposta; e no Capítulo 5, apresenta-se a conclusão deste trabalho, além de breve discussão sobre as limitações da abordagem escolhida e possíveis linhas de trabalho que poderão ser seguidas no futuro.

# Capítulo 2

# Referencial teórico

## 2.1 Body Sensor Network (BSN)

A Body Sensor Network (BSN) é um sistema de monitoramento médico. Seu projeto inicial foi expandido ao decorrer dos anos a partir de múltiplos projetos que foram desenvolvidos pelos participantes do Laboratório de Engenharia de Software da Universidade de Brasília (LES-UnB). O objetivo do projeto é criar um sistema que é auto-adaptativo em relação à sua confiabilidade.

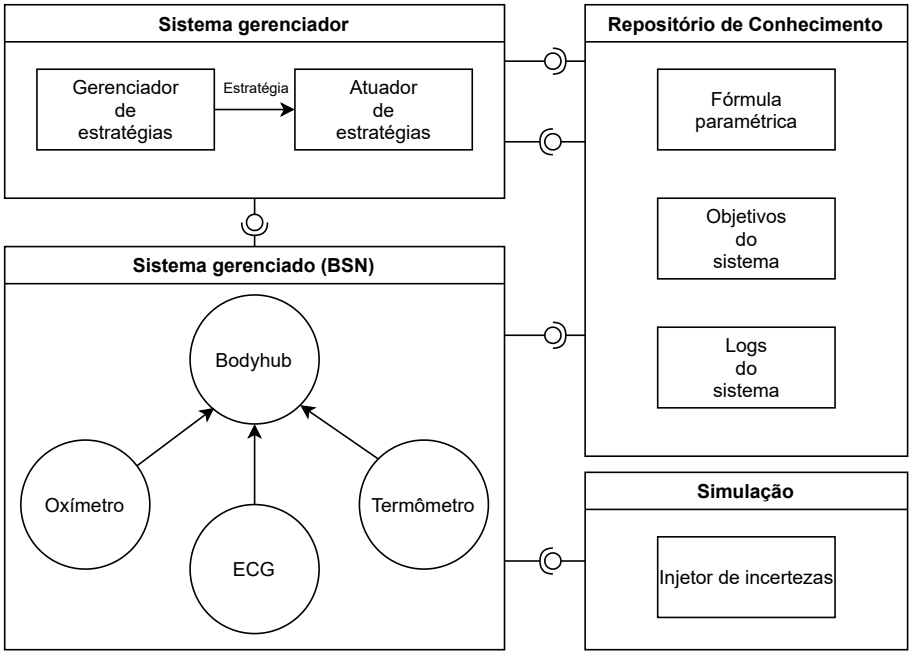


Figura 2.1: Representação arquitetural da BSN [1]

A Figura 2.1 consiste numa representação arquitetural[1] da BSN. Sua arquitetura atualmente segue a VerAACiTy, um modelo de verificação de sistemas auto-adaptativos, dividido em camadas. As camadas podem ser descritas brevemente como sendo:

- Sistema gerenciador: camada responsável pela definição e aplicação de estratégias de auto-adaptação com base no cálculo de confiabilidade do sistema;
- Sistema gerenciado: camada que comporta o sistema o qual está sendo monitorado pelo gerenciador. Seu monitoramento é realizado por meio dos componentes *probe* e *effector*, que realizam as ações de coletar dados do sistema gerenciado e aplicar ações com base nas estratégias definidas pelo controlador, respectivamente. Neste caso, o sistema gerenciado em questão é a própria BSN;
- Repositório de conhecimento: camada que contém informações necessárias para o cálculo de confiabilidade, como a fórmula paramétrica que é utilizada e os dados que serão utilizados no cálculo, dentre outras coisas;
- Simulação: camada responsável por exercitar o modelo por meio de incertezas que são aplicadas, a partir da simulação de ruídos presentes em contextos que seriam necessários que o sistema se adapte.

Na parte do sistema gerenciado, foco deste trabalho, cada círculo corresponde a um componente seu. As setas, por sua vez, indicam como é feita sua integração, o fluxo de dados que é seguido. O sistema funciona a partir da coleta de dados de um paciente, por meio de sensores (e.g., termômetro, oxímetro, eletrocardiograma, etc.) e o seu processamento por meio de um componente centralizador de informações, denominado *bodyhub* (ou centralhub). O resultado desse processo é o estado de saúde do paciente, que pode variar entre baixo, médio ou alto risco.

Por conta de sua natureza médica, esse sistema pode ser considerado um sistema crítico. A possibilidade de perda de dados importantes que são coletados, além de outros fatores, podem acabar resultando em um risco para a vida do paciente, dependendo de suas condições de saúde prévias (e.g., falha ao aferir a pressão de um paciente que possui hipertensão).

Dessa forma, um fator importante a ser considerado ao desenvolver esse sistema é sua *confiabilidade*. O sistema precisa se assegurar de sua disponibilidade, além de que ele opera de forma correta. Para tanto, a BSN foi projetada e implementada como um sistema auto-adaptativo, o que significa que, entre outras características principais, possui conhecimento sobre seus próprios objetivos. Considerando que um desses objetivos para a BSN consiste em manter sua confiabilidade em um nível aceitável, isso acaba sendo utilizado como o objetivo para monitorar se o sistema está, de fato, funcionando devidamente.

A Figura 2.2 ilustra os requisitos da BSN utilizando um modelo de objetivos contextual (ou *Contextual Goal Model*, como visto em outras produções do grupo). Esse tipo de modelo descreve como o comportamento do sistema pode ser dividido em objetivos, que são decompostos em tarefas, até que atinjam tarefas-folha, que são tarefas que já não podem

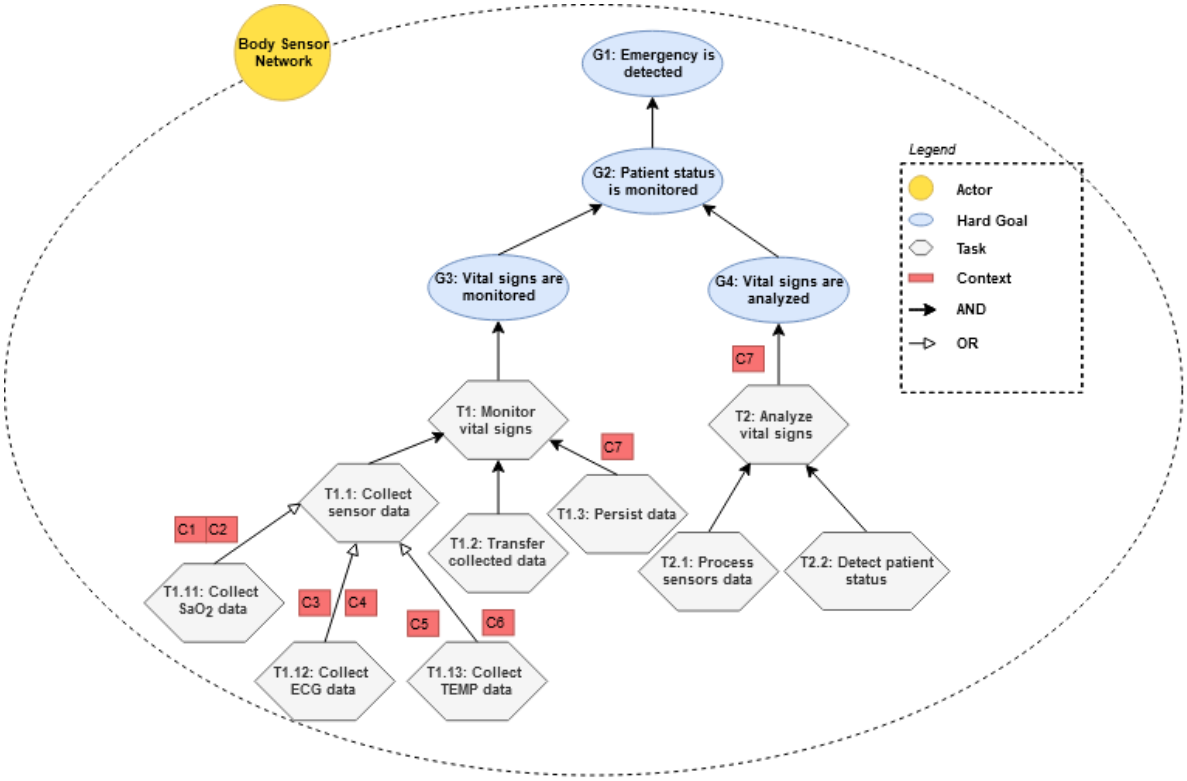


Figura 2.2: Modelo de objetivos contextual da BSN[2] adaptado para três sensores

ser mais decompostas em outras. Existem dois tipos de decomposição: *decomposição-OR* e *decomposição-AND*, sendo que cada uma representa seu equivalente lógico. Isso significa que um objetivo decomposto com decomposição-AND em duas outras tarefas deve possuir ambas as tarefas completas para que seja considerado alcançado; já um objetivo decomposto com decomposição-OR em outras duas tarefas, basta que uma das tarefas esteja concluída para que seja considerado alcançado. Os retângulos vermelhos representam contextos os quais o sistema poderá estar sujeito, sendo eles:

- C1: Oxímetro (SPO2) está disponível
- C2: Dados coletados pelo SPO2 se encontram em um intervalo válido
- C3: Eletrocardiograma (ECG) está disponível
- C4: Dados coletados pelo ECG se encontram em um intervalo válido
- C5: Termômetro (TEMP) está disponível
- C6: Dados coletados pelo TEMP se encontram em um intervalo válido
- C7: Nível de bateria do *Bodyhub*

A BSN pode ser considerada como sendo um sistema ciber-físico, o que significa que é um sistema que possui interação com o meio físico, assim como é o caso de robôs,

por exemplo. Com isso, ela acaba por herdar vantagens e desvantagens de tais tipos de sistema, sendo um exemplo de desvantagem a dificuldade na hora de planejar. Isso se dá, entre outros motivos, justamente por conta de seu envolvimento com o meio físico, pois essa característica implica em existir uma variabilidade grande nos contextos nos quais esse tipo de sistema poderá estar sujeito. Dado isso, garantir que tal sistema está cumprindo propriedades desejáveis com relação a critérios como sua confiabilidade, por exemplo, se torna uma tarefa trabalhosa e nada prática.

Além de ser necessário verificar todos os cenários aos quais dado sistema poderia estar sujeito, ainda há a necessidade de certo conhecimento que só pode ser obtido durante o tempo de execução, o que dificulta o trabalho de projeto do sistema durante sua fase de *design*. Sua implementação foi feita inicialmente utilizando o *framework* OpenDaVINCI[10], porém posteriormente, adotou-se o *framework* Robot Operating System(ROS) por possuir um suporte maior, assim como por sua ampla utilização na implementação de sistemas desse tipo.

Durante os anos, alguns esforços foram feitos para tentar assegurar que algumas de suas propriedades estivessem em devido funcionamento. Dentre essas abordagens, cita-se a que culminou no artigo submetido ao SEAMS[10], uma das principais conferências da área.

Esse artigo, dentre outros resultados apresentados, evidenciou um método criado que utilizava checagem de modelos, coleta de dados *on-line* e mineração de dados para assegurar que determinadas propriedades funcionais e não funcionais fossem alcançadas. Isso foi um grande marco para o grupo e após isso, uma série de trabalhos relacionados à checagem de modelos foi realizado. Uma das ramificações desses trabalhos resultou na VerAACiTY[8], que é o modelo arquitetural apresentado no início desta seção. Para evidenciar a funcionalidade do arcabouço arquitetural que foi criado, a BSN foi utilizada como um estudo de caso.

## 2.2 Robot Operating System (ROS)

O *Robot Operating System (ROS)* [11] é um conjunto de bibliotecas e ferramentas utilizado para criar software de cunho robótico. Consiste em um *framework open-source* flexível projetado pela organização *Open Robotics*, com uma grande comunidade ativa na internet. Ele possibilita a criação de programas que utilizam componentes denominados **nós**, que são executados dentro de um ambiente e se comunicam entre si a partir de mensagens de um protocolo *publisher and subscriber* (ou "publicador e inscrito", em tradução livre).

Esse protocolo, por sua vez, permite que um nó publique uma mensagem (geralmente consistindo de um valor que é calculado utilizando algum algoritmo) em um determi-

nado tópico, enquanto outros nós se inscrevem nesse tópico, com o objetivo de recuperar mensagens, para que dessa forma possam utilizá-las para outras tarefas.

O conteúdo das mensagens trocadas entre os nós é definido pelo programador. Como documentado em sua *wiki*, o ROS permite uma grande variedade de tipos de dados para compô-las. Dessarte, as ferramentas providas pelo *framework* podem ser utilizadas para a criação de diversos tipos de sistemas, com tamanhos dos mais variados.

A BSN, por possuir uma natureza distribuída, beneficia-se grandemente dessas ferramentas providas, especialmente por questões de abstração do sistema. Utilizando-se da noção de nós, cada componente da BSN é implementado como um nó auto-contido, que computa seus próprios dados (e.g., temperatura do paciente, nível de oxigenação, etc.) e os publica como mensagens.

Dentre algumas das vantagens de utilizar esse *framework*, destacam-se:

- Amplo conjunto de ferramentas para testar e simular cenários para os sistemas criados, o que possibilita a criação de *softwares* mais robustos;
- Comunidade ativa, seja por fóruns ou até mesmo no desenvolvimento do projeto *open-source*;
- Sua abstração de processos em forma de nós auto-contidos possibilita uma múltipla escolha de linguagens de forma relativamente simples, permitindo implementar nós tanto em *Python* quanto em *C++*, com uma certa interoperabilidade;
- Utilização de arquivos contendo configurações que podem ser resgatadas em tempo de execução permitem uma certa flexibilidade na hora de implementar.

## 2.3 Compreensão de programa

Compreensão de programa é um processo cognitivo importante para o desenvolvimento de sistemas [12]. Desenvolvedores de software passam boa parte de seu tempo realizando o processo de entendimento de um sistema, seja por meio de análise de código-fonte, ferramentas de *debug*, etc. Mesmo com mais de 30 anos de pesquisa no campo de engenharia de software, esse tempo não diminuiu de forma notável [12]. Uma das razões que contribuem para esse fenômeno é a falta de progresso nas pesquisas de *compreensão de programa* nos anos 90, que duraram cerca de dez anos.

Portanto, por mais que exista um bom volume de ferramentas que são capazes de ajudar um desenvolvedor nessa tarefa, a maioria delas não foi avaliada suficientemente. Existe uma série de abordagens utilizadas já estudadas para aumentar a questão da compreensão, dentre elas, a abordagem *top-down* de modelagem de compreensão de programa

é utilizada quando desenvolvedores já estão familiarizados com o domínio da aplicação, porém não possuem noções mais específicas em relação aos detalhes de implementação [13].

Essa abordagem utiliza a ideia de uma hipótese de natureza geral do programa, que então é refinada ao ponto de que esses desenvolvedores poderão, eventualmente, possuir uma noção mais baixo-nível sobre o programa. Dessa forma, é possível avaliar, modificar, aceitar ou rejeitar as hipóteses resultantes desse refinamento.

A teoria de compreensão *bottom-up* de compreensão de programa, em contraste com a *top-down*, assume que programadores leem primeiro os comandos do código, e após isso agrupam mentalmente esses comandos, com objetivo de alcançar abstrações de nível cada vez maior. Repete-se esse processo em abstrações cada vez maiores, até que se obtenha um entendimento alto-nível do programa.

Classifica-se o tipo de conhecimento possuído de um programa entre sintático e semântico, sendo o primeiro tipo dependente da linguagem e relacionado com os comandos e com as unidades básicas em um programa, enquanto que o outro é um conhecimento que independe de detalhes específicos, como a linguagem, e é construído em camadas progressivas até que um modelo mental capaz de descrever o domínio da aplicação seja formado.

Um modelo desse tipo geralmente é desenvolvido a partir de uma abstração inicial de fluxo de controle, que captura a sequência de operações num programa [12]. Esse modelo é o *modelo do programa*, e é desenvolvido pelo agrupamento de microestruturas no texto (comandos, fluxo de controle e relacionamentos) em macroestruturas (abstrações de estruturas de texto) junto com referências cruzadas desas estruturas. Assim que o modelo de programa for totalmente assimilado, um *modelo situacional* é desenvolvido. Esse modelo junta conhecimentos sobre abstrações de fluxo de dados e abstrações funcionais.

Além dos dois tipos de abordagem apresentadas, ainda existem uma série de outras, que muitas vezes se utilizam de combinações de ambas, ou até mesmo de misturas entre alguns elementos de cada, com o objetivo de gerar uma visão mais clara do programa como um todo. Um exemplo disso é a abordagem de metamodelo integrada, que possui quatro componentes principais, utilizando-se de modelos *top-down*, de programa, de fluxo de dados e de uma base de conhecimentos adquiridos e inferidos para gerar o entendimento final sobre o programa ao desenvolvedor necessitado dessa informação [12].

## 2.4 UPPAAL

O UPPAAL é uma ferramenta de modelagem, simulação e verificação de sistemas em tempo real desenvolvida a partir de uma colaboração entre o Departamento de Tecnologia da Informação da Uppsala University, na Suécia, e o Departamento de Ciência da

Computação da Universidade de Aalborg, na Dinamarca. Tal ferramenta integra modelagem, validação e verificação de sistemas em tempo real modelados como redes de autômatos temporais, estendido com tipos de dados (inteiros, *arrays*, etc.) [5, 6]. Utiliza-se, como mencionado, de autômatos temporais, além de notação *Computation Tree Logic (CTL)* [14] para a definição de propriedades de sistema que podem ser verificadas em nível de modelo de sistema.

Essas propriedades são úteis para testar a corretude de um modelo, que pode ser posteriormente utilizado como referência para a implementação de um dado sistema, utilizando-se de uma abordagem baseada em modelos (*model-based approach*).

Em UPPAAL as propriedades TCTL (*timed-CTL*) podem ser expressas conforme descrito na Tabela 2.1

| **Expressão** | **Semântica** |
|---|---|
| E$\Diamond$ $\phi$ | existe um caminho onde $\phi$ eventualmente ocorrerá |
| A$\Box$ $\phi$ | para todos os caminhos $\phi$ sempre ocorre |
| E$\Box$ $\phi$ | existe um caminho onde $\phi$ sempre ocorre |
| A$\Diamond$ $\phi$ | para todos os caminhos $\phi$ eventualmente ocorrerá |
| $\phi \rightarrow \psi$ | quando $\phi$ ocorre, $\psi$ eventualmente ocorrerá |
| $\phi$ imply c $\leq$ T | $\phi$ ocorre se e somente se o relógio c está dentro de T |
| A$\Box$ not deadlock | verifica existência de *deadlocks* |

Tabela 2.1: Expressões básicas da CTL no UPPAAL

A partir da gramática apresentada na Tabela 2.1, é possível expressar uma série de propriedades úteis para os sistemas em tempo real, incluindo, mas não limitadas a: *liveness*, *reachability* e *safety*.

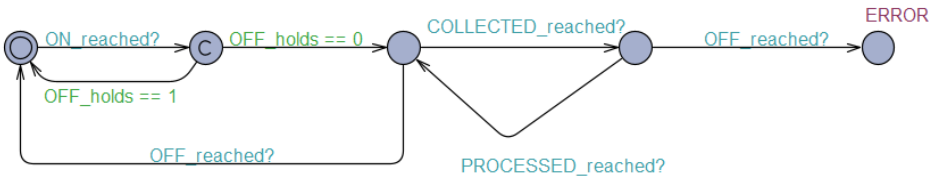


Figura 2.3: Exemplo de um modelo UPPAAL

A Figura 2.3 mostra um autômato temporal modelado no UPPAAL consistindo no modelo do monitor da seguinte propriedade TCTL em UPPAAL:

$$sensornode.collected \rightarrow bodyhub.processed \tag{2.1}$$

Os elementos visuais presentes na Figura 2.3 são informalmente explicados abaixo:

- Círculos: representam os estados do autômato. O círculo com um outro círculo dentro representa o estado inicial, já o com a letra "C"representa um estado *commited*, que significa um estado que deve ser avaliado no momento em que se chega nele, durante uma simulação;
- Setas: representam as transições entre estados;
- Transições com interrogação: canais que representam mensagens que o autômato está esperando receber de outro para que transitem;
- Transições com operador de igualdade: são as guardas da transição, utilizadas para garantir que a transição só ocorrerá quando a condição for satisfeita.
- ERROR: estado final do autômato que caracteriza o fim da execução com erro.

Os modelos de autômatos temporais e propriedades em TCTL a seguir, juntamente com o modelo da figura 2.3, foram utilizados como inspiração para a implementação dos nós monitores de propriedade deste trabalho. Destacam-se os devidos créditos de autoria a Marc Carwehl, integrante do *Software Engineering Group* da *Humboldt-Universität zu Berlin*, Alemanha.

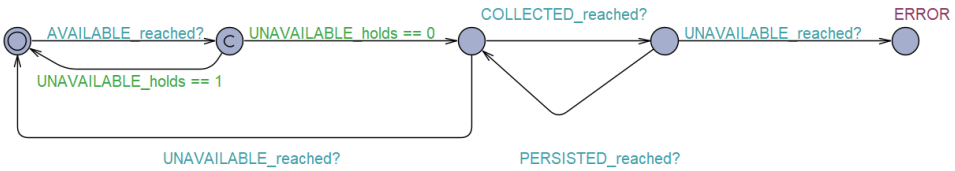


Figura 2.4: Modelo UPPAAL para o monitor da propriedade P2

A Figura 2.4 mostra um autômato temporal modelado no UPPAAL consistindo no modelo do monitor da seguinte propriedade TCTL em UPPAAL:

$$sensornode.collected \rightarrow bodyhub.persisted \tag{2.2}$$

A Figura 2.5 mostra um autômato temporal modelado no UPPAAL consistindo no modelo do monitor da seguinte propriedade TCTL em UPPAAL:

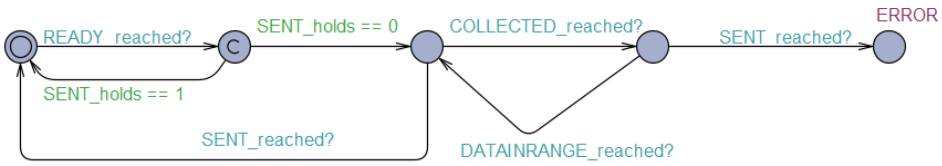


Figura 2.5: Modelo UPPAAL para o monitor da propriedade P3

$$sensornode.collected \rightarrow sensornode.inrange \tag{2.3}$$

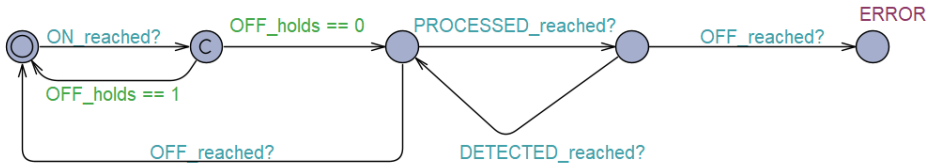


Figura 2.6: Modelo UPPAAL para o monitor da propriedade P4

A Figura 2.6 mostra um autômato temporal modelado no UPPAAL consistindo no modelo do monitor da seguinte propriedade TCTL em UPPAAL:

$$bodyhub.processed \rightarrow bodyhub.detected \tag{2.4}$$

# Capítulo 3

# Metodologia

## 3.1 Definição das propriedades desejáveis

A metodologia utilizada neste trabalho consiste no esforço de utilizar o ROS, juntamente com algumas noções de compreensão de programa, para implementar novos nós que monitoram propriedades na BSN durante sua execução. Fazendo isso, é possível verificar a viabilidade da implementação de monitores em tempo real para este sistema.

Foi utilizada uma abordagem baseada em modelos para esta implementação, partindo de propriedades modeladas previamente no UPPAAL (*c.f.* Seção 2.4), em autômatos temporais e CTL, como base para estruturar os novos nós que realizam o monitoramento.

### 3.1.1 Propriedade P1

A propriedade P1 foi baseada no autômato temporal da Figura 2.3 (ou na fórmula TCTL 2.1), e pode ser descrita como: "Entre o sensor estar ligado ou desligado, em resposta aos dados coletados pelo sensor, estes serão eventualmente processados pelo *bodyhub*". Essa propriedade é utilizada para testar que cada sinal vital monitorado por um determinado sensor será processada pelo *bodyhub*.

Do ponto de vista do funcionamento do sistema, é algo importante para ser testado, dado o fato de que todo sinal vital, ou pelo menos uma boa parte, deve ser processado para que o *bodyhub* possa apresentar um estado de saúde preciso do usuário. A propriedade falha (entra no estado ERROR) nos casos em que um sinal vital é coletado pelo sensor, mas o sensor desliga antes de que possa mandar o dado. Quando isso acontece, o *bodyhub* já não é capaz de realizar o processamento dos dados, eventualmente causando um erro de cálculo da condição do paciente.

Os diagramas de sequência para a propriedade P1 estão representados nas Figuras 3.1 e 3.2, onde são mostrados os casos em que a mesma é satisfeita e não satisfeita, respecti-

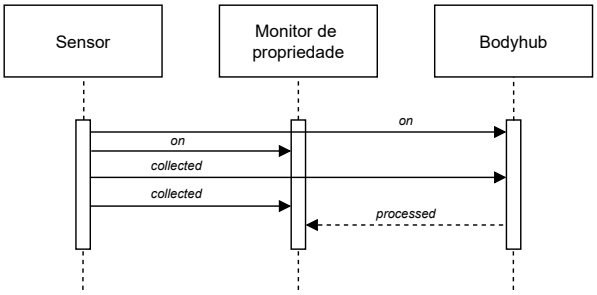


Figura 3.1: Diagrama de sequência para o caso em que a propriedade P1 é satisfeita

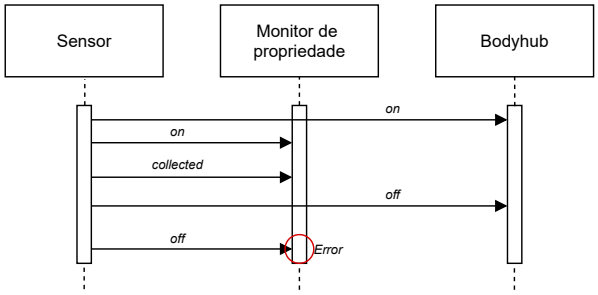


Figura 3.2: Diagrama de sequência para o caso em que a propriedade P1 não é satisfeita

vamente. Para esses diagramas, assim como os presentes nas outras subseções, buscou-se utilizar a notação mais próxima possível do que foi utilizado nos modelos UPPAAL utilizados como base para este projeto. A interação entre os atores, representadas pelas setas, fica ainda mais fácil de entender se aliada à Figura 3.9, dado que fica evidente a forma com a qual os atores interagem entre si. Destaca-se que as mensagens enviadas pelos sensores aos outros atores, por exemplo, *on* e *collected*, são o mesmo dado sendo enviado via mensagem para dois tópicos diferentes, e não dois dados diferentes coletados em momentos distintos. Isso também é válido para as outras propriedades.

### 3.1.2 Propriedade P2

A propriedade P2 foi baseada no autômato temporal da Figura 2.4 (ou na fórmula TCTL 2.2), e consiste na afirmação: "No período em que o sensor está ligado e há uma coleta de dados da parte do sensor, o *bodyhub* irá eventualmente persistir o dado em resposta". Essa propriedade é utilizada para testar que, para cada sinal vital monitorado por um sensor, este será persistido pelo *bodyhub*.

A persistência é um passo essencial para o bom funcionamento do algoritmo do *bodyhub*, pois o processamento dos dados para o cálculo do estado de um paciente depende diretamente de dados que foram previamente persistidos. Portanto, é essencial que os dados persistidos sejam de fato bem recebidos.

Essa propriedade falha se algum sensor por algum motivo se torna indisponível após coletar o dado, porém antes de que o *bodyhub* possa persisti-lo, causando uma perda de informação, e, consequentemente, um cálculo impreciso do resultado final do risco de saúde do paciente.

Os diagramas de sequência para a propriedade P2 estão representados nas Figuras 3.3 e 3.2, onde são mostrados os casos em que a mesma é satisfeita e não satisfeita, respectivamente. A última figura é a mesma da apresentada na propriedade P1, pois

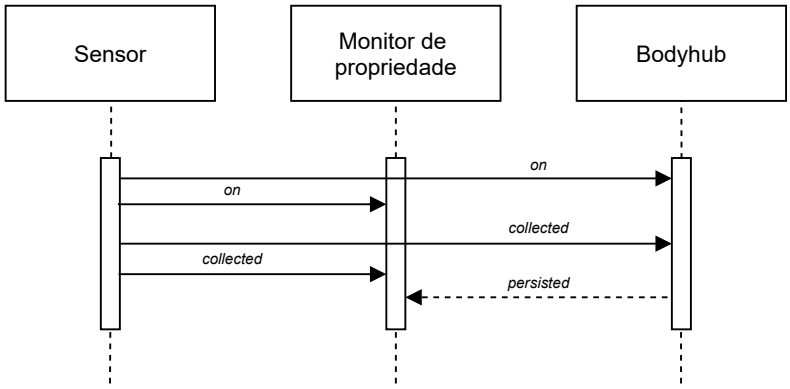


Figura 3.3: Diagrama de sequência para o caso em que a propriedade P2 é satisfeita

para este trabalho, os contextos em que um sensor se encontra indisponível resumem-se ao caso em que ele está ligado além disso, como é possível notar por essas figuras, do ponto de vista do diagrama de sequência, a propriedade P2 é similar à propriedade P1, pois os atores são os mesmos (ambas as propriedades envolvem ações vindas do sensor e do *bodyhub*). Apesar disso, as interações trocadas entre eles são diferentes. Para que o monitor de propriedade indique uma falha, é necessário que a mensagem que receba logo após o *collected* do sensor seja um *off* do mesmo sensor, indicando que este coletou um dado, porém desligou antes que pudesse enviá-lo para o *bodyhub*, consequentemente não o permitindo que fosse eventualmente persistido, o que viola a propriedade, como foi descrita.

### 3.1.3 Propriedade P3

A propriedade P3 foi baseada no autômato temporal da Figura 2.5 (ou pela fórmula TCTL 2.3), e consiste na afirmação: "Entre o sensor estar pronto e enviar seus dados, em resposta à coleta de dados, o sensor garantirá que estes estão dentro do intervalo". Essa propriedade é relacionada a duas atividades dos sensores: coleta de dados e checagem do dado (para verificar se o mesmo se encontra dentro de um intervalo que faça sentido). Ela falha no caso em que o sensor manda dados que não se encontram dentro do intervalo, resultando em possíveis cálculos errados no caso em que esse dado for eventualmente processado pelo *bodyhub*.

Os diagramas de sequência para a propriedade P3 estão representados nas Figuras 3.4 e 3.5, onde são mostrados os casos em que a mesma é satisfeita e não satisfeita, respectivamente. Para o caso de sucesso, o diagrama representa que o sensor deve, entre estar pronto para coletar, e enviar seus dados (representado pelas mensagens *ready* e *sent*, respectivamente), é necessário que ele garanta que estes estejam dentro do intervalo considerado correto pelo sensor, o que é representado pela mensagem *in_range*.

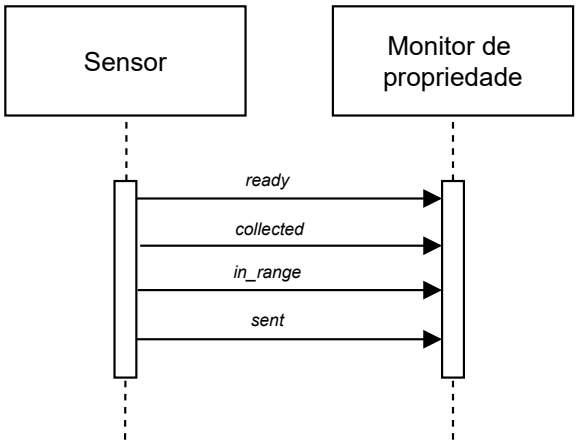


Figura 3.4: Diagrama de sequência para o caso em que a propriedade P3 é satisfeita

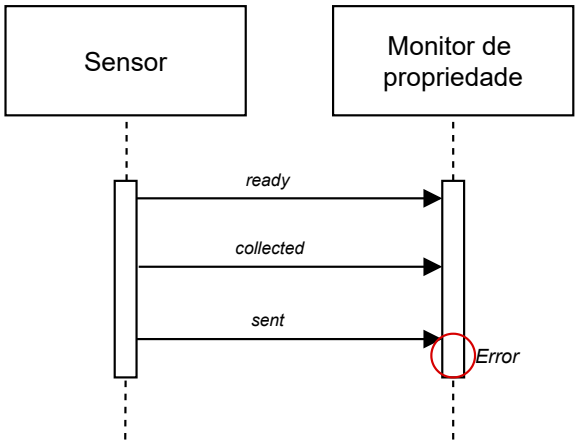


Figura 3.5: Diagrama de sequência para o caso em que a propriedade P3 não é satisfeita

Para o caso de falha, destaca-se que o diagrama não possui a mensagem *in_range*, ou seja, o sensor enviou os seus dados sem que viesse a verificar sua corretude antes de fazê-lo, o que é considerada uma violação da propriedade P3.

### 3.1.4 Propriedade P4

A propriedade P4 foi baseada no autômato temporal da Figura 2.6 (ou pela fórmula TCTL 2.4), e consiste na afirmação: "Enquanto o *bodyhub* estiver ligado, em resposta ao seu processamento de dados, ele irá eventualmente detectar um novo estado de saúde de um paciente". Essa propriedade é relacionada com duas atividades realizadas pelo *bodyhub*: processamento e detecção. Ela falha no caso em que o *bodyhub* é capaz de processar determinado dado, mas não detecta o estado de saúde do paciente.

Os diagramas de sequência para a propriedade P4 estão representados nas Figuras 3.6 e 3.7, onde são mostrados os casos em que a mesma é satisfeita e não satisfeita, respectivamente. Para o caso de sucesso, o diagrama mostra o *bodyhub* indicando que está ligado, processando dados recebidos pelos sensores e, por fim, detectando o estado de saúde do paciente, cumprindo o que é proposto na propriedade P4. Para o caso de falha, o diagrama mostra o caso em que o *bodyhub* desliga (representado pela mensagem *off*) antes que consiga realizar a detecção do estado de saúde do paciente, violando o que foi proposto pela propriedade.

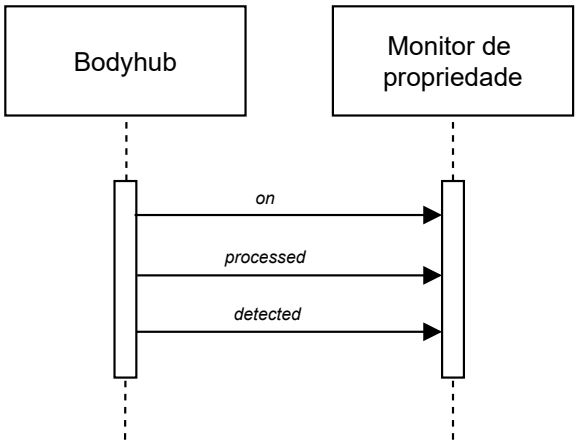


Figura 3.6: Diagrama de sequência para o caso em que a propriedade P4 é satisfeita

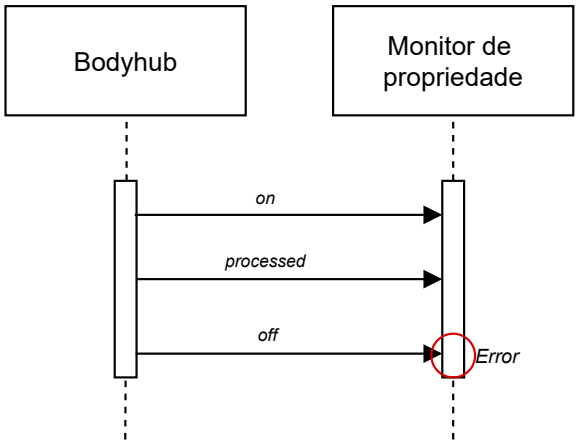


Figura 3.7: Diagrama de sequência para o caso em que a propriedade P4 não é satisfeita

## 3.2 Utilização de compreensão de programa

Até chegar no ponto em que se encontra no momento do projeto dos novos nós, a BSN passou por uma série de mudanças. Essas mudanças são dos mais variados tipos, sejam mudanças de adições de novos sensores, alterações de código básicas, até mesmo chegando a mudar o *framework* que era utilizado, o OpenDaVINCI, para o ROS.

Além disso, diversas mudanças arquiteturais foram realizadas, especialmente se considerar que a versão que se encontra disponível hoje é a que foi utilizada como estudo de caso para a implementação da VerAACiTy[8], o que culminou na adição de diversos nós que não eram relacionados ao sistema em si, mas a um arcabouço muito mais amplo, envolvendo até mesmo a adição de uma série de componentes relacionados à verificação de parâmetros da teoria de controle. Ademais, boa parte dos nomes dos componentes foi retirada diretamente do artigo, o que para alguém leigo sobre o que foi feito naquele trabalho, seria ainda mais difícil de entender.

Dessarte, por conta das proporções que o projeto tomou, já não é mais uma tarefa simples para os desenvolvedores do grupo de pesquisas (ou para outros desenvolvedores não tão familiarizados com o sistema) expandirem o trabalho feito sem que um tempo considerável seja investido para o entendimento de como cada componente executa suas tarefas e se comunica com os outros. Futuramente, isso pode acabar vindo a desencorajar possíveis extensões desse trabalho, o que não é algo desejado pelo grupo de pesquisa, visto que vários projetos já foram concluídos com sucesso, com diversos artigos aceitos em algumas revistas reconhecidas na área.

Para o caso deste trabalho, essa dificuldade também se mostrou um impeditivo, especialmente nos primeiros contatos com o projeto. Apesar de diversas reuniões e docu-

mentações já existentes (especialmente por meio de artigos publicados), relacionar o que muitas vezes estava mais claro na literatura com o que acontecia no sistema, em um nível mais "baixo", tornou-se uma tarefa complicada sem meios sistematizados para alcançar tal entendimento.

Para tentar reduzir essa dificuldade, um trabalho de iniciação científica que levava em conta elementos da compreensão de programas foi realizado, utilizando a BSN como seu estudo de caso. A partir de resultados gerados desse trabalho, utilizando certos aspectos de uma metodologia *top-down* de compreensão, em que se explora o projeto a partir de certo conhecimento sobre seu funcionamento geral, evoluindo para conhecimentos mais específicos, dada a investigação de pontos específicos do código e da formulação de hipóteses que, posteriormente seriam validadas, foi possível adquirir um conhecimento mais aprofundado sobre o projeto para que, este trabalho, de fato, fosse realizado.

Primeiramente, buscou-se entender de forma geral sobre o que era o projeto. Após isso, buscou-se gerar visões sobre esse projeto, estática e dinâmica. A primeira permite ao programador entender relações que são formadas dentro do projeto, a partir de hierarquia de classes, composições, implementações de interfaces, entre outras coisas. Uma visão dinâmica, por sua vez, permite ao programador ver, de uma forma mais inteligível, como que cada componente se comunica com outro durante a execução do projeto como um todo.

Existem ferramentas que já são amplamente utilizadas para auxiliar na geração de cada uma dessas visões. Para a geração de visão estática, uma ferramenta de documentação automática, como o Doxygen, pode ser de grande ajuda. Dado que o Doxygen documenta o projeto, discriminando suas classes e a relação entre cada uma delas, é possível verificar possíveis candidatos do projeto que devem ser estendidos, certas variáveis que devem ser utilizadas em um componente ROS, entre outras possíveis deduções.

Para a geração de uma visão dinâmica, por sua vez, optou-se por utilizar a ferramenta *rqt_graph*, própria da instalação completa do ROS, pois ela permite que se tenha uma visão geral de como os nós estão interagindo durante o tempo de execução, incluindo os tópicos os quais estão sendo publicados e inscritos, em determinado momento, de forma amigável ao usuário. A partir dessa visão, é possível verificar pontos em que seria possível incluir um novo nó, ou até mesmo instrumentar para a realização de testes, dentre outras coisas.

Para a realização desse trabalho, buscou-se gerar ambas as visões estática e dinâmica, visto que seria interessante ter noções tanto do ponto de vista da arquitetura de código, quanto da arquitetura de componentes em tempo de execução, para entender como e onde poderiam ser criados os novos nós necessários. A última visão foi de especial importância, pois como boa parte das propriedades testadas incluem dados provenientes de sensores e do

*bodyhub*, é necessário entender onde que esses dados poderiam ser adquiridos para utilizar como a entrada dos novos nós, dando uma noção de um fluxo de dados, representados por mensagens trocadas em tempo de execução.

### 3.2.1 Visão estática da BSN

A geração de visões estáticas para um programa é importante para se ter aumentar a compreensão sobre forma com a qual dado programa está implementado. Esse tipo de visão é gerada a partir do código-fonte de uma aplicação, e produz como resultado informações que podem ser detectadas numa análise estática de código, como é feito por diversos softwares. Para o caso da BSN, buscou-se utilizar o programa Doxygen, que, apesar de ser mais conhecido por ser utilizado na geração automática de documentação de programas, possui um módulo que se integra ao GraphViz para a geração de grafos que representam algumas visões estáticas importantes, como a de chamada de métodos, grafos de dependências, entre outros tipos.

A Figura 3.8 é o grafo de chamadas de métodos da classe *Sensor*, classe que é utilizada na representação de nós sensores dentro da BSN. Além da geração dessa visão, visto que se tratava de um arquivo com extensão *SVG (Scalable Vector Graphics)*, que é um subconjunto de arquivos de extensão *XML (eXtended Markup Language)* utilizada para a representação de imagens, foi criado um programa para colorir as entidades do programa, de forma que cada uma delas representa o domínio de cada um dos métodos.

Isso foi interessante, pois pôde mostrar que existe uma complexa relação de hierarquia de classes dentro do programa. Além disso, para a instrumentação deste projeto, foi importante ter essa visão, pois é possível saber em qual nível de abstração da classe que representa o nó é necessário se colocar determinada publicação ou inscrição. Um exemplo disso é a presença dos métodos *Sensor::TurnOn()* e *Sensor::TurnOff()*, que possuem relação com algumas propriedades que vieram a ser monitoradas, como a P1 e P4, fazendo com que o entendimento do nível da hierarquia de classes fosse essencial para saber onde enviar as mensagens que representam cada uma dessas informações.

Os domínios os quais cada método pode pertencer, juntamente com sua explicação, são os descritos a seguir:

- Libbsn: biblioteca de diversas classes e métodos utilizada para apoiar o funcionamento do sistema como um todo. Dentre alguns de seus módulos, destacam-se o módulo responsável por representar o modelo de objetivos (possui abstrações de objetivo, tarefa, tarefa-folha, etc.), e o módulo responsável pela representação da cadeia de Markov, utilizada para a geração aleatória de dados;

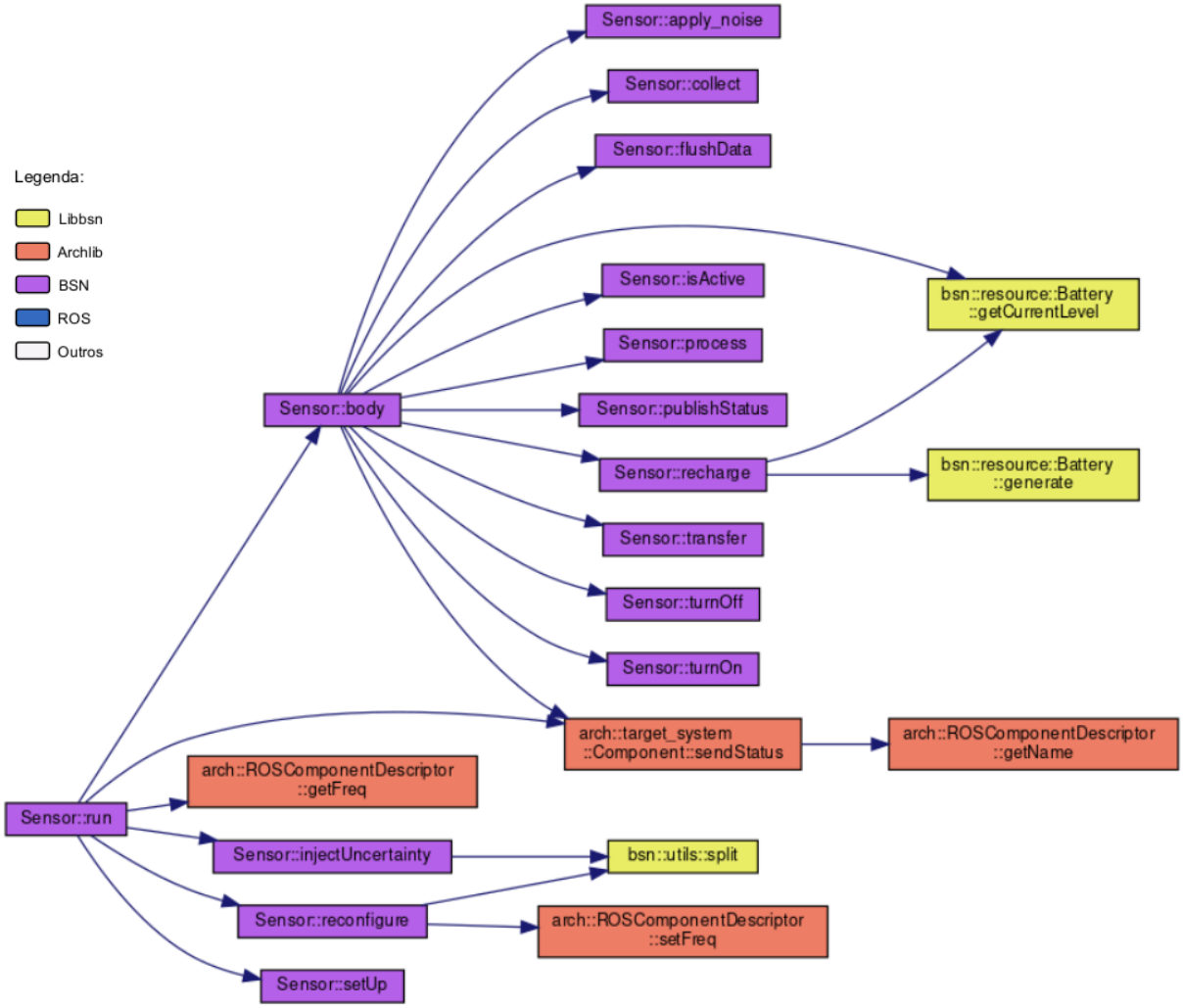


Figura 3.8: Grafo de chamada de métodos de um sensor da BSN, gerado pelo Doxygen

- Archlib: biblioteca que contém a definição de classes gerais utilizadas na implementação da arquitetura da BSN. Dentre alguns de seus módulos, destaca-se o *ROSComponentDescriptor*, que é uma classe da qual todos os nós pertencentes à arquitetura herdam, possuindo um comportamento geral que é seguido por cada um de seus filhos;
- BSN: módulo que possui a implementação das classes utilizadas na implementação de fato da BSN. Possui, portanto, famílias de classes que representam sensores, *bodyhub*, dentre outros módulos pertencentes à implementação da BSN como se encontra hoje;
- ROS: módulo que representa os métodos e componentes próprios do *framework* ROS;
- Outros: artefatos gerais, presentes na própria linguagem C++.

### 3.2.2 Visão dinâmica da BSN

A geração de visões dinâmicas é importante do ponto de vista de compreensão de programa, pois, por meio delas é possível identificar interações entre entes pertencentes a um programa, durante seu tempo de execução. A Figura 3.9 mostra uma visão dinâmica da BSN, capturada durante seu tempo de execução, juntamente com nós monitores de propriedades, adição deste trabalho.

Cada elipse representa um nó dentro do ambiente ROS. As setas que chegam a um nó representam o tópicos nos quais aquele nó está inscrito em dado momento, enquanto que as setas que saem de um nó representam os tópicos nos quais o nó está publicando naquele momento. Assim, se existe uma seta entre dois nós em dado momento, existe comunicação entre eles.

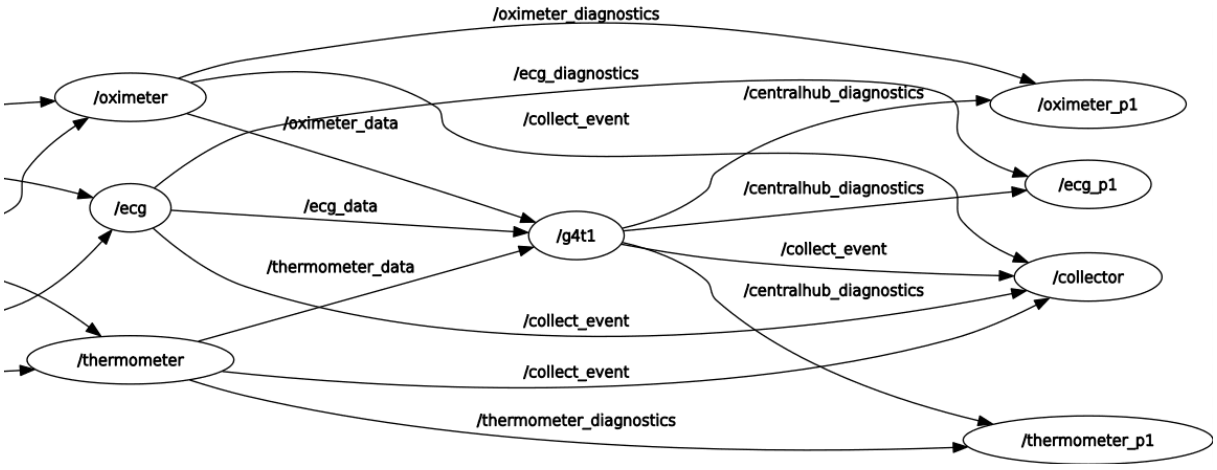


Figura 3.9: Visão dinâmica dos nós da BSN durante execução

Os nós adicionados decorrentes da implementação realizada neste trabalho são *oximeter_p1, ecg_p1* e *thermometer_p1*, os tópicos adicionados foram: *oximeter_diagnostics, ecg_diagnostics, thermometer_diagnostics* e *centralhub_diagnostics*. Note que, para cada nó relacionado ao monitoramento de propriedade (nós com o sufixo *p1*), existem duas setas que chegam a este, que são justamente as entradas dos sensores e do *bodyhub* (também chamado de centralhub ou g4t1 nesse esquema).

Nos tópicos com sufixo *diagnostics*, trafegam mensagens cuja estrutura utiliza de dados que já são utilizados no funcionamento padrão da BSN, porém adicionados de alguns mecanismos que permitem fácil identificação por meio dos nós monitores (utilização de números identificadores, informação sobre destinatário, etc.)

Os dados coletados relativos aos valores mensurados pelos sensores, por sua vez, não são pertinentes à verificação das propriedades selecionadas e, portanto, não são utilizados nas mensagens em si. Por meio da adição desses novos nós e tópicos, possibilita-se que sejam realizados os experimentos planejados.

É possível traçar um paralelo desta visão com a visão arquitetural apresentada na Figura 2.1, pois cada um dos sensores, além do *bodyhub* (g4t1, nesse caso) encontram-se presentes, além de suas comunicações. Isso reforça a ideia de que a visão dinâmica pode ser de grande proveito para programadores e até mesmo arquitetos de *software* que desejam criar um sistema interconectados.

## 3.3 Artefato de solução proposto

O artefato computacional proposto neste trabalho como solução para o problema de monitoramento é uma versão adaptada da BSN acrescida de nós monitores. Estes também são nós ROS que são executados juntamente com a BSN para captar e analisar os dados que são enviados via mensagens durante a execução.

A análise realizada corresponde a cada propriedade disponibilizada no modelo UPPAAL (e.g., o nó monitor para propriedade P1 analisa dados que são coletados por um sensor e processados pelo *bodyhub*). Cada sensor da BSN, além do *bodyhub*, será propriamente instrumentado para mandar cópias de seus dados para novos tópicos, via novas mensagens. Esses tópicos, consequentemente, serão utilizados pelos monitores implementados para verificar se as propriedades estão sendo satisfeitas enquanto a BSN é executada.

As mensagens serão alteradas para incluir identificadores únicos, com o intuito de verificar se determinada mensagem que é enviada é a mesma que é verificada depois. Isso serve como uma forma de garantir a ordem das mensagens, enquanto se mantém a abstração de um autômato. Cada monitor foi feito baseado em seu autômato observador correspondente. Dessa forma, boa parte da programação envolvida foi derivada de lógica intrínseca a máquinas de estado (i.e., o estado de um processo muda de acordo com determinada entrada).

A Figura 3.10 mostra uma breve visão de como os novos nós, que fazem o papel de monitores, estão inseridos com a versão adaptada da BSN utilizada neste trabalho, exemplificado para o caso de P1. Os nós adicionados servem para monitorar continuamente os fluxos de dados de cada componente do sistema gerenciado (sensores e *bodyhub*).

Para realizar tal feito, os sensores em si tiveram sua implementação alterada com o intuito de adicionar mecanismos que permitissem que essas mensagens pudessem ser recebidas por esses nós, o que foi feito por meio da criação de tópicos novos, para os quais essas mensagens fossem enviadas além do tópico para o qual já estavam sendo originalmente. O *bodyhub* também foi alterado, de forma que parte da mensagem original também é enviada em forma de uma nova mensagem para este tópico que é utilizado para o monitoramento.

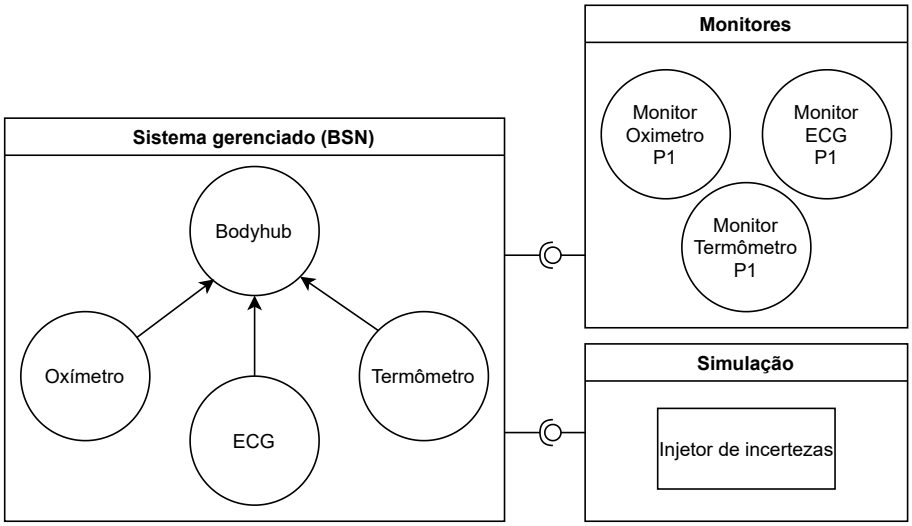


Figura 3.10: Fluxo de monitoramento

Os nós monitores, por sua vez, consomem as mensagens enviadas para cada um desses tópicos novos, gerando suas análises de propriedades a partir deles. Note que uma relação mais detalhada de como eles interagem entre si (tópicos utilizados, nome dos nós em tempo de execução, etc.) pode ser verificada na visão dinâmica da Figura 3.9

Cada propriedade acaba equivalendo a pelo menos um nó adicional sendo executado juntamente ao sistema, de forma que este é associado aos componentes que estão sendo monitorados em determinada propriedade. Isso significa que, para cada execução, para as propriedades P1, P2 ou P3, existem um total de três nós adicionais em tempo de execução, enquanto que, para a propriedade P4, apenas um nó adicional, visto que para as primeiras existe a necessidade de monitorar a saída de cada sensor, enquanto para a última, apenas o monitoramento do *bodyhub* já é suficiente.

Para cada um deles, buscou-se basear sua implementação ao autômato que foi originalmente criado para cada propriedade. A versão da BSN apresentada no artefato proposto consiste de uma versão reduzida em relação à BSN utilizada em outros projetos, contando com um total de três sensores, em contraste com o que geralmente está presente nos outros trabalhos relacionados a ela (geralmente com cinco sensores associados). O motivo para tal redução se trata de algo técnico, que será posteriormente discutido, na seção de limitações do projeto.

O ROS, como mencionado previamente, utiliza um protocolo *publisher and subscriber*, que dita como componentes do sistema trocam dados entre si, denominadas mensagens, a partir da publicação e inscrição em tópicos que utilizam uma estrutura de dados de fila. Para processar essa fila, o ROS utiliza um método chamado *spin*, que é utilizado para processar funções de *callback*, que consistem de funções implementadas dentro de um nó, associadas ao comportamento de processamento de mensagens provenientes de

determinado tópico.

Durante a implementação de um nó, esse método geralmente é inserido ao fim do método principal de um nó. Dentro de um nó, para cada tópico em que se encontra inscrito, existe uma função de *callback* associada a ele, o que significa que o ROS possui uma forma padrão de lidar com sua fila de mensagens, o que pode causar certa limitação quando se trata da representação de autômatos.

Um autômato possui o comportamento de esperar por determinada entrada para mudar seu estado interno. Dessa forma, deve existir alguma forma de representar a lógica dessa espera. Para tanto, buscou-se utilizar uma variação bastante utilizada do método *spin*, mencionado anteriormente, o *spinOnce*, que permite que o comportamento do nó entre cada execução sua seja mais detalhado. Com essa variação, é possível adquirir o primeiro elemento da fila de mensagens e processá-lo como uma entrada para o monitor, permitindo que o comportamento pertinente aos autômatos mencionado acima possa ser simulado.

Para a implementação de fato, utilizou-se a noção do fluxo principal do autômato, com cada um de seus estados representados por um *loop* constante, onde cada estado depende da ativação de certas variáveis *booleanas*. Dessa forma, enquanto tais variáveis (ou *flags*) não são ativadas, o programa permanece no mesmo estado, simulando o que ocorreria no caso de um autômato que espera uma entrada para transitar. As *flags* são ativadas no fluxo principal, logo após o processamento de um método de *callback*, representando a chegada de um valor na entrada para ser processado. Busca-se então, por meio das *flags* e da iteração principal, realizar o papel de se assemelhar a um autômato.

Existem, porém, alguns desafios associados a essa abordagem, pois assume-se que cada tópico de mensagens possua um tamanho próprio, para prevenir a possibilidade da perda de dados durante a execução do sistema e, consequentemente, resultar em um monitoramento faltoso. Existe também a possibilidade de algumas mensagens serem perdidas por conta de problemas de concorrência, o que pode se remediado a partir da mudança de frequências nas quais cada componente será executado.

Também é necessário assegurar-se de que as mensagens não seriam reenviadas aos nós monitores, com o *bodyhub* sendo o maior desafio para realizar tal feito por conta de sua frequência, que deve ser elevada em relação aos outros sensores, para que possa acomodar um número grande de mensagens chegando (considerando que recebe mensagens de todos os sensores), que pode acabar resultando em um número maior de mensagens sendo enviadas ao monitor, o que, por sua vez, pode resultar em certas propriedades monitoradas constando como satisfeitas, mesmo em casos que não são.

Foi necessário também realizar alguns passos adicionais por conta de detalhes mais inerentes às especificidades de implementação, o que já era de se esperar, pois isso já é

levado em consideração quando se implementa algo a partir de um modelo.

Visto que há certa imprevisibilidade no fluxo principal do programa, devido principalmente à sua forma de utilizar métodos de *callback* e, como mencionado, o *spin*, fica difícil garantir que o programa, ao voltar para esse fluxo (que implementa o que mais se assemelha do fluxo do autômato), possua as *flags* necessárias para a transição de estado. Isso ocorre principalmente por conta da ordem e origem das mensagens, pois para o *spin*, não há discriminação do tipo e origem da mensagem que será processada, incorrendo na perda de detecção de alguns erros. Esse é um problema que foi contornado a partir da verificação de "remetente"da mensagens dentro dos métodos de *callback*, ao invés do fluxo principal.

Para realizar esses experimentos, é importante mencionar que todos os componentes da BSN, como também os nós monitores adicionados em sua versão do artefato proposto, produzem arquivos de *log*, que contém os dados que estão sendo trocados em determinado momento, os erros detectados, além do tempo em que esses dados estão sendo medidos, para uma noção de temporalidade. Na próxima seção, será descrita em mais detalhes como esses arquivos foram utilizados para produzir as análises.

```
2021-10-28T02:55:38.783344,15,thermometer,on
2021-10-28T02:55:49.469976,15,thermometer,collected
2021-10-28T02:55:49.470028,15,thermometer,sensor accuracy fail
2021-10-28T02:55:49.474061,16,thermometer,collected
2021-10-28T02:55:49.474127,16,thermometer,sent
2021-10-28T02:55:50.476315,17,thermometer,collected
2021-10-28T02:55:50.476586,17,thermometer,sent
2021-10-28T02:55:51.474349,18,thermometer,collected
2021-10-28T02:55:51.474606,18,thermometer,sensor accuracy fail
2021-10-28T02:55:52.477105,19,thermometer,collected
2021-10-28T02:55:52.477462,19,thermometer,sent
2021-10-28T02:55:53.474829,20,thermometer,collected
2021-10-28T02:55:53.475164,20,thermometer,sent
2021-10-28T02:55:54.470289,21,thermometer,off
```

Figura 3.11: Exemplo de *log* produzido pelo termômetro

A Figura 3.11 mostra a estrutura de um arquivo de *log* persistido pelo termômetro. Essa estrutura consiste de:

- *Timestamp* do momento de registro do dado (seja coleta ou envio);
- Número identificador (*id*) da mensagem;
- Sensor de origem (neste caso, o termômetro);
- A informação sobre a mensagem (se foi coletada, enviada, se o sensor ligou ou desligou).

Como é possível notar, o sensor, por conta própria, já possui mecanismo para saber se o dado coletado se encontra ou não correto, pois este já possui verificação própria. A

quantidade elevada de falhas se dá por conta da injeção de ruídos, que para o Experimento 2, foi configurado para ser mais frequente que o usual.

```
2021-10-28T02:55:38.783344,15,thermometer,on
2021-10-28T02:55:49.469976,15,thermometer,collected
2021-10-28T02:55:49.470028,15,thermometer,sensor accuracy fail
2021-10-28T02:55:49.474061,16,thermometer,collected
2021-10-28T02:55:49.494968,16,thermometer,processed
2021-10-28T02:55:50.476315,17,thermometer,collected
2021-10-28T02:55:51.474349,18,thermometer,collected
2021-10-28T02:55:51.474606,18,thermometer,sensor accuracy fail
2021-10-28T02:55:52.477105,19,thermometer,collected
2021-10-28T02:55:52.496829,19,thermometer,processed
2021-10-28T02:55:53.474829,20,thermometer,collected
2021-10-28T02:55:54.470289,21,thermometer,off
ERROR: 2021-10-28T02:55:54.470289
```

Figura 3.12: Exemplo de *log* produzido pelo nó monitor *(observer)*

A Figura 3.12 mostra a estrutura de um arquivo de *log* que é persistido pelo nó monitor durante a verificação da propriedade P1. Note que, em contraste com a Figura 3.11, existem informações de erro adicionais presentes na estrutura. Esses erros adicionados são justamente os erros inéditos detectados pelo monitor, que seriam impossíveis ao próprio sensor detectar por conta própria. Eles acontecem justamente nos pontos em que o sensor coleta, porém desliga antes de enviar, o que é justamente a descrição da propriedade P1, que é a que está sendo monitorada neste caso.

```
2021-10-28T02:55:28.884328,14,thermometer,processed
2021-10-28T02:55:28.884526,14,thermometer,detected
2021-10-28T02:55:28.884679,14,thermometer,persisted
2021-10-28T02:55:49.494968,16,thermometer,processed
2021-10-28T02:55:49.495031,16,thermometer,detected
2021-10-28T02:55:49.495065,16,thermometer,persisted
2021-10-28T02:55:50.495654,16,oximeter,processed
2021-10-28T02:55:50.495823,16,oximeter,detected
2021-10-28T02:55:50.496056,16,oximeter,persisted
2021-10-28T02:55:52.496829,19,thermometer,processed
2021-10-28T02:55:52.497019,19,thermometer,detected
2021-10-28T02:55:52.497162,19,thermometer,persisted
```

Figura 3.13: Exemplo de *log* produzido pelo *bodyhub*

Note a relação que há entre os dados que são registrados nos *logs*. Para os exemplos selecionados nas Figuras 3.11, 3.12 e 3.13, é possível verificar que elementos identificadores das mensagens que são enviadas pelos sensores e pelo *bodyhub*, como as *timestamps*, o *id* e a informação do nó, são os mesmos presentes no *log* do monitor. Isso acontece por conta de que, apesar de serem enviadas mensagens diferentes, os dados enviados são os mesmos. A seleção em vermelho nas figuras representam mensagens enviadas pelos sensores, enquanto que as a em azul, mensagens enviadas pelo *bodyhub*.

# Capítulo 4

# Resultados

## 4.1 Projeto dos experimentos

Para validar o artefato proposto, foram planejados três experimentos:

- O primeiro experimento consiste na utilização de nós com *mock* de dados para verificar os caminhos descritos por cada um dos diagramas presentes no Capítulo 3. Busca-se por meio deste experimento exercitar cada um dos caminhos para os nós monitores que foram implementados neste trabalho, com o objetivo de evidenciar seu bom funcionamento.

  Para realizar tal feito, serão instrumentados nós que enviarão as mensagens específicas para alcançar cada um dos casos, de sucesso e de falha, para cada uma das propriedades.

- O segundo experimento consiste em uma avaliação qualitativa com respeito à possibilidade e à utilidade da implementação de nós monitores dentro da BSN. Para essa análise, buscou-se executar o sistema 10 vezes, com execuções de 5 minutos, enquanto um nó à parte, responsável pela injeção de ruídos, injetava ruídos propositalmente danosos aos sensores, causando algumas aferições faltosas.

  O ruído foi configurado para se comportar como uma curva quadrada, com amplitude de 0.3 e frequência de 0.25 Hz.

  Após isso, compara-se a média de erros detectados apenas pelo sistema (sem os monitores) durante sua execução com a média erros detectados pelos monitores, o que foi feito a partir de uma comparação dos *logs* produzidos pelos componentes próprios da BSN (sensores e *bodyhub*) com os *logs* produzidos pelos monitores.

  Se os nós monitores forem capazes de detectar mais erros do que o sistema por conta própria, considera-se que o artefato é, de fato, útil para o caso da BSN.

- Para o terceiro experimento, foi utilizado o programa Stacer, que é uma ferramenta de monitoramento de recursos em tempo de execução para ambientes *Unix-like*. Essa ferramenta permite que sejam monitorados recursos de uso de memória, em megabytes, como também uso de CPU, em percentuais.

  O programa permite o monitoramento de processos de forma individual, o que torna possível detectar os processos relacionados especificamente ao monitoramento, permitindo adquirir uma métrica final de consumo para cada recurso durante o tempo de execução da BSN.

  Para cada propriedade monitorada, o artefato proposto foi executado 5 vezes durante 2 minutos, permitindo, então, a coleta de 5 amostras de cada recurso computacional, uma a cada 24 segundos. Após isso, calcula-se o valor médio e desvio padrão com o objetivo de buscar considerações sobre o uso de recursos.

Os experimentos foram executados em um computador com processador Intel Core I5-9600KF @ 3,60GHz x 6, com 16GB de memória RAM, executando o sistema operacional Ubuntu versão 20.04.3 LTS.

## 4.2 Resultados obtidos

### 4.2.1 Resultados do Experimento 1

Para o Experimento 1, buscou-se utilizar o nó de monitor de propriedades como uma espécie de caixa-preta. Dessa forma, foi possível instrumentar sua entrada de maneira que, para determinadas entradas, esperou-se um comportamento bem específico. Assim, tendo em vista os diagramas de sequência presentes no Capítulo 3, foi possível saber a ordem das mensagens que devem ser enviadas, para o caso de cada propriedade, de forma que seja considerado um sucesso ou uma falha.

Dessarte, evidenciou-se que os monitores cumpriram sua função de monitoramento. Em detalhes de implementação, o nó de *mock* de dados publicou para o tópico do oxímetro e do *bodyhub*, vide Figura 3.9. Apenas o oxímetro foi utilizado, visto que os outros sensores possuem comportamento idêntico do ponto de vista de implementação.

Para a propriedade P1, para evidenciar o caminho de sucesso, instrumentou-se o *mock* de forma que:

- Publica *on* no tópico *oximeter_diagnostics*;
- Publica *collected* no tópico *oximeter_diagnostics*;
- Publica *processed* no tópico *centralhub_diagnostics*;

- Publica *off* no tópico *oximeter_diagnostics*;

Já para evidenciar o caminho de falha para P1, instrumentou-se o *mock* de forma que a seguinte sequência de mensagens fosse enviada:

- Publica *on* no tópico *oximeter_diagnostics*;
- Publica *collected* no tópico *oximeter_diagnostics*;
- Publica *off* no tópico *oximeter_diagnostics*;

As Figuras 4.1 e 4.2 mostraram os *logs* gerados das execuções de cada uma das instrumentações do *mock* mencionadas acima, para caso de sucesso e de falha da propriedade P1, respectivamente.

```
2021-10-27T04:09:49.851088,1,oximeter,on
2021-10-27T04:09:50.184355,1,oximeter,collected
2021-10-27T04:09:50.517729,1,oximeter,processed
2021-10-27T04:09:50.851025,1,oximeter,off
2021-10-27T04:09:51.184353,2,oximeter,on
2021-10-27T04:09:51.517741,2,oximeter,collected
2021-10-27T04:09:51.851080,2,oximeter,processed
2021-10-27T04:09:52.184408,2,oximeter,off
```

Figura 4.1: Arquivo de *log* para o caso de sucesso de P1

```
2021-10-27T04:11:18.675633,1,oximeter,on
2021-10-27T04:11:19.008965,1,oximeter,collected
2021-10-27T04:11:19.342265,1,oximeter,off
ERROR: 2021-10-27T04:11:19.342265
2021-10-27T04:11:19.675635,2,oximeter,on
2021-10-27T04:11:20.008966,2,oximeter,collected
2021-10-27T04:11:20.342265,2,oximeter,off
ERROR: 2021-10-27T04:11:20.342265
```

Figura 4.2: Arquivo de *log* para o caso de falha de P1

Para a propriedade P2, para evidenciar o caminho de sucesso, instrumentou-se o *mock* de forma que:

- Publica *on* no tópico *oximeter_diagnostics*;
- Publica *collected* no tópico *oximeter_diagnostics*;
- Publica *persisted* no tópico *centralhub_diagnostics*;
- Publica *off* no tópico *oximeter_diagnostics*;

Já para evidenciar o caminho de falha para P2, instrumentou-se o *mock* de forma que a seguinte sequência de mensagens fosse enviada:

- Publica *on* no tópico *oximeter_diagnostics*;
- Publica *collected* no tópico *oximeter_diagnostics*;
- Publica *off* no tópico *oximeter_diagnostics*;

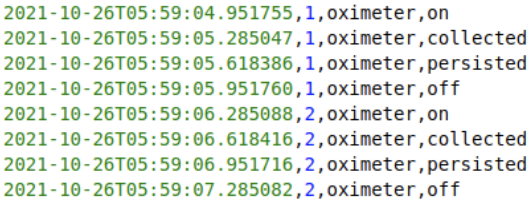


Figura 4.3: Arquivo de *log* para o caso de sucesso de P2

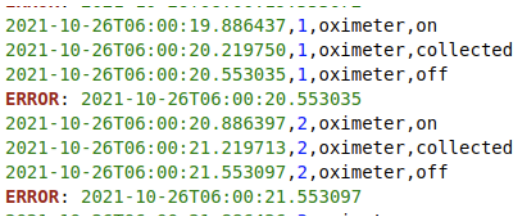


Figura 4.4: Arquivo de *log* para o caso de falha de P2

As Figuras 4.3 e 4.4 mostraram os *logs* gerados das execuções de cada uma das instrumentações do *mock* mencionadas acima, para caso de sucesso e de falha da propriedade P2, respectivamente.

Para a propriedade P3, para evidenciar o caminho de sucesso, instrumentou-se o *mock* de forma que:

- Publica *ready* no tópico *oximeter_diagnostics*;
- Publica *collected* no tópico *oximeter_diagnostics*;
- Publica *data in range* no tópico *sensor_inrange*;
- Publica *sent* no tópico *oximeter_diagnostics*;

Já para evidenciar o caminho de falha para P3, instrumentou-se o *mock* de forma que a seguinte sequência de mensagens fosse enviada:

- Publica *ready* no tópico *oximeter_diagnostics*;
- Publica *collected* no tópico *oximeter_diagnostics*;
- Publica *sent* no tópico *oximeter_diagnostics*;

As Figuras 4.5 e 4.6 mostraram os *logs* gerados das execuções de cada uma das instrumentações do *mock* mencionadas acima, para caso de sucesso e de falha da propriedade P3, respectivamente.

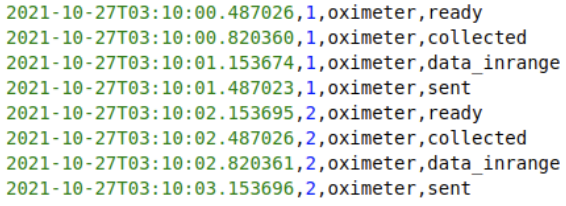


Figura 4.5: Arquivo de *log* para o caso de sucesso de P3

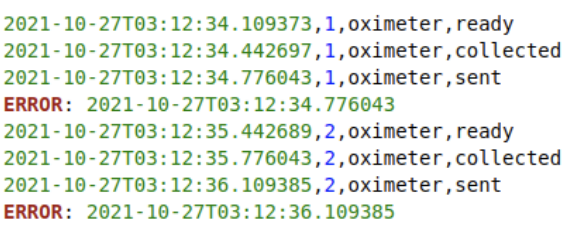


Figura 4.6: Arquivo de *log* para o caso de falha de P3

Para a propriedade P4, para evidenciar o caminho de sucesso, instrumentou-se o *mock* de forma que:

- Publica *on* no tópico *centralhub_diagnostics*;
- Publica *processed* no tópico *centralhub_diagnostics*;
- Publica *detected* no tópico *centralhub_detected*;
- Publica *off* no tópico *centralhub_diagnostics*;

Já para evidenciar o caminho de falha para P4, instrumentou-se o *mock* de forma que a seguinte sequência de mensagens fosse enviada:

- Publica *on* no tópico *centralhub_diagnostics*;
- Publica *processed* no tópico *centralhub_diagnostics*;
- Publica *off* no tópico *centralhub_diagnostics*;

As Figuras 4.7 e 4.8 mostraram os *logs* gerados das execuções de cada uma das instrumentações do *mock* mencionadas acima, para caso de sucesso e de falha da propriedade P4, respectivamente.

```
2021-10-27T04:07:13.192289,1,oximeter,on
2021-10-27T04:07:13.525621,1,oximeter,processed
2021-10-27T04:07:13.858932,1,oximeter,detected
2021-10-27T04:07:14.192273,1,oximeter,off
2021-10-27T04:07:14.525588,2,oximeter,on
2021-10-27T04:07:14.858952,2,oximeter,processed
2021-10-27T04:07:15.192250,2,oximeter,detected
2021-10-27T04:07:15.525591,2,oximeter,off
```

Figura 4.7: Arquivo de *log* para o caso de sucesso de P4

```
2021-10-27T03:58:15.268815,1,oximeter,on
2021-10-27T03:58:15.602150,1,oximeter,processed
2021-10-27T03:58:15.935550,1,oximeter,off
ERROR: 2021-10-27T03:58:15.935550
2021-10-27T03:58:16.268792,2,oximeter,on
2021-10-27T03:58:16.602182,2,oximeter,processed
2021-10-27T03:58:16.935476,2,oximeter,off
ERROR: 2021-10-27T03:58:16.935476
```

Figura 4.8: Arquivo de *log* para o caso de falha de P4

### 4.2.2 Resultados do Experimento 2

Para o Experimento 2, buscou-se coletar 10 amostras de valores obtidos de cada execução da BSN com os nós monitores. Cada execução durou 5 minutos.

Para garantir o tempo de execução correto, foram criados scripts *bash* contendo a chamada de execução de cada nó, além do comando utilizado para se encerrar a execução de todos os nós do ambiente, o *rosnode kill -a*. Os resultados obtidos estão discriminados abaixo.

Para a propriedade P1, os seguintes resultados foram obtidos durante a execução do experimento:

| P1 | Oxímetro | ECG | Termômetro |
|---|---|---|---|
| Média de mensagens enviadas | 105,5 ± 2,01 | 104,4 ± 2,50 | 107,3 ± 1,33 |
| Erros detectados sem monitor | 48,8 ± 3,15 | 41 ± 5,96 | 42,9 ± 5,66 |
| Erros detectados com monitor | 63,7 ± 3,16 | 54,8 ± 5,61 | 57,3 ± 5,33 |
| Aumento (%) | 30,53 | 33,65 | 33,56 |

Tabela 4.1: Resultados do Experimento 2 para a propriedade P1

Com essas informações, calculou-se média e desvio padrão para os valores de aumento para o caso de P1, resultando nas seguintes informações:

- Média do aumento de detecção dos erros: 32,58%;
- Desvio padrão do aumento de detecção dos erros: 1,77%.

Isso mostrou que, para a propriedade P1, houve um aumento significativo de detecção de erros de sistema por parte dos nós monitores, especialmente se levou em conta de que esses erros são erros que, caso não monitorados, poderiam vir a causar possíveis problemas futuros, especialmente por se tratar de um sistema crítico.

| P2 | Oxímetro | ECG | Termômetro |
|---|---|---|---|
| Média de mensagens enviadas | 104,6 ± 2,67 | 103 ± 2,90 | 104,9 ± 3,57 |
| Erros detectados sem monitor | 45,2 ± 8,23 | 41,3 ± 4,11 | 43,5 ± 2,46 |
| Erros detectados com monitor | 60 ± 7,78 | 56,3 ± 5,71 | 57,7 ± 2,49 |
| Aumento (%) | 32,74 | 36,31 | 32,64 |

Tabela 4.2: Resultados do Experimento 2 para a propriedade P2

Com essas informações, calculou-se média e desvio padrão para os valores de aumento para o caso de P2, resultando nas seguintes informações:

- Média do aumento de detecção dos erros: 33,9%;
- Desvio padrão do aumento de detecção dos erros: 2,21%.

A propriedade P2, por sua vez, acompanhou a tendência da propriedade P1, mostrando um aumento no valor número de detecções de erro bem parecido.

Para a propriedade P3, buscou-se fazer uma alteração no comportamento dos sensores, pois estes possuem naturalmente o comportamento de descartar valores que são medidos e se encontram fora dos limites preestabelecidos.

Alterou-se para que esse valor não fosse mais descartado, tendo em vista que a propriedade verifica justamente a questão da possibilidade de uma fonte externa (nó monitor) ser capaz de verificar a corretude do dado. Fazendo isso, considera-se que os sensores já

não são mais capazes de recalcular seus dados no caso de perceberem essa falha em sua mensuração, passando a enviá-los mesmo no caso em que estão errados.

Tendo isso em vista, é mostrado abaixo o resultado das médias e desvios padrão calculados nas execuções de P3:

- Média de mensagens enviadas pelo oxímetro: 100,5 ± 2,59
- Média de erros detectados no oxímetro pelo nó monitor: 47,3 ± 4,11
- Média de mensagens enviadas pelo termômetro: 100 ± 2,82
- Média de erros detectados no termômetro pelo nó monitor: 38,8 ± 3,29
- Média de mensagens enviadas pelo ECG: 100,1 ± 2,96
- Média de erros detectados no ECG pelo nó monitor: 50,7 ± 9,64

Logo, considerando a alteração feita nos sensores, notou-se um aumento muito desejável da detecção de erros. Isso mostra a importância da existência de monitores externos ao sistema, por exemplo, no caso em que não foi possível que os componentes verificassem suas próprias falhas.

Para a propriedade P4, por se tratar de uma propriedade que trata de estados relacionados ao *bodyhub*, instrumentou-se o componente de forma que esse pudesse enviar mensagens ao monitor ao receber uma mensagem (para representar o momento em que entra em estado de processamento), e logo depois de detectar o estado do paciente, utilizando seus cálculos específicos para isso. Com isso, expôs-se um estado do sistema que anteriormente, com o objetivo de torná-lo observável e analisável.

Além disso, alterou-se a configuração de bateria do sensor do *bodyhub*, de forma que ele gastou energia ao executar uma detecção, dado que a falha dessa propriedade se deu entre as tarefas de processamento e detecção.

O *bodyhub* é um nó que por si só não possui uma verificação de erros, fazendo com que o nó monitor seja desejável para a detecção de erros. Os seguintes resultados foram obtidos:

- Quantidade de mensagens processadas: 94 ± 1,41;
- Quantidades de erros detectados pelo monitor: 5,8 ± 2,28.

Assim, pelo fato do *bodyhub* não possuir uma forma interna de verificação de falhas, - pois assumiu-se que sensores já possuem em sua implementação aquele comportamento de descartar seus dados fora de intervalo, fazendo com que uma dupla verificação seja apenas para motivos de redundância - concluiu-se que a utilização de um nó monitor para verificar a propriedade P4 seria desejável, para que erros possam ser observados e possivelmente evitados no futuro.

### 4.2.3 Resultados do Experimento 3

Para o terceiro experimento, utilizou-se o software *Stacer* para medir os valores de consumo de recursos da BSN, com e sem os sensores de monitoramento que foram adicionados, com o intuito de possuir uma comparação e realizar uma breve análise sobre a viabilidade da implementação no que tangem esses recursos.

Coletaram-se os dados conforme o que foi colocado no início deste capítulo, ou seja, coletas de amostras a cada 24 segundos de execução, considerando o valor final de dado recurso para a execução como sendo a média dessas amostras, tendo em vista atenuar disparidades que ocorrem, especialmente por conta da primeira amostra, que costuma vir com valores maiores por se tratar de um estágio de execução perto do inicial, mostrando um consumo razoavelmente maior que as demais.

A escolha da ferramenta *Stacer* para a verificação de tais valores deveu-se por conta principalmente da facilidade de verificar o consumo de cada recurso de forma separada, permitindo ainda que sejam filtrados os processos, o que facilitou a coleta, que foi feita de forma manual.

Para questões de cálculo, utilizou-se o consumo de recursos da BSN de forma separada (sem os monitores e nós correlatos). Com isso, considerou-se valores dos nós que executam os processos principais de cada componente, além de valores de consumo de processos genéricos do ROS, visto que, em condições normais, esses valores seriam executados e, portanto, contabilizando para o cálculo final do consumo médio de recursos.

Foram obtidas as seguintes informações:

- Média e desvio padrão de consumo de memória RAM da BSN (em *megabytes*): 610,4353 ± 0.3841;
- Média e desvio padrão de consumo percentual de CPU da BSN: 18,27143 ± 1,7623;

Os valores de consumo total de recursos para cada propriedade, incluindo valores de consumo da BSN e dos monitores, são descritos a seguir:

- P1: memória(MB): 806,96 ± 0,53; CPU(%): 21,90 ± 0,58
- P2: memória(MB): 807,04 ± 0,55; CPU(%): 22,19 ± 0,61
- P3: memória(MB): 807,19 ± 0,48; CPU(%): 22,19 ± 0,53
- P4: memória(MB): 676,23 ± 0,55; CPU(%): 19,83 ± 0,48

Esses valores, por sua vez, são utilizados como base para o cálculo da Tabela 4.3, que mostra a proporção de aumento de uso de cada um dos recursos, para cada uma das propriedades.

| Propriedade | Mem (%) | CPU (%) |
|---|---|---|
| P1 | 32,19 ± 0,03 | 16,77 ± 0,97 |
| P2 | 32,18 ± 0,03 | 18,35 ± 0,99 |
| P3 | 32,2 ± 0,03 | 18,32 ± 0,41 |
| P4 | 10,75 ± 0,01 | 5,41 ± 0,83 |

Tabela 4.3: Tabela de resultados do Experimento 3

A partir dessas informações, nota-se que o monitoramento, da forma que é proposta no artefato de solução deste trabalho, é custoso.

Considerando-se que no máximo três nós são monitorados por vez, com exceção da propriedade P4, pode se considerar que o acréscimo de memória RAM para cada nó monitor é de mais de 10%, enquanto que, em termos de consumo de CPU, é mais de 5% para a mesma quantidade.

Para a versão da BSN que foi utilizada neste trabalho, que conta com apenas três de sensores, e por se tratar de uma simulação de *software*, esse custo ainda é viável. Porém, transpor esse custo para uma implementação embarcada, por exemplo, ainda precisa-se avaliar sua viabilidade.

# Capítulo 5

# Conclusões

## 5.1 Conclusões gerais

Conclui-se este trabalho respondendo as questões propostas na seção de introdução. De fato, é possível implementar nós que realizam a tarefa de monitorar constantemente a BSN, com o objetivo de verificar propriedades previamente modeladas, enquanto se preserva sua arquitetura. Como explicado, essa tarefa pode ser realizada a partir da instrumentação dos nós já existentes, com o objetivo de tornar certos estados observáveis, para que possam ser monitorados por componentes externos.

Além disso, foi visto que é útil realizar tal feito, pelo motivo de que esses monitores podem não só detectar uma quantidade maior de erros, mas como também detectar erros que jamais seriam detectados pelo próprio sistema. Dessa forma, verificou-se que esse monitoramento é um processo custoso.

Considerando que, mesmo para o caso de monitoramentos mais básicos, como *debugs*, adicionam-se algumas camadas de complexidade, o acréscimo de fatores em tempo real e suas complexidades intrínsecas tornam essa solução útil, porém com um custo elevado.

A BSN, como foi mencionado durante este trabalho, é um sistema que possui uma natureza embarcada (mesmo que, no atual momento, se encontre apenas em um estágio de simulação em *software*.) Isso significa que, por mais que durante este estágio, ela possa se beneficiar de uma quantidade mais abundante de recursos computacionais, como uma quantidade maior de memória RAM, ou até mesmo um processador de desempenho mais elevado, caso sejam consideradas as limitações de *hardware* características de sistemas embarcados, essa solução pode simplesmente não ser viável para este caso específico.

Por mais que exista a possibilidade de que se adicione mais capacidade de *hardware*, essa opção pode ser custosa financeiramente. Apesar disso, não significa que a ideia de utilizar monitoramento em tempo real deva ser completamente descartada.

Dessarte, reforça-se a ideia de que este projeto foi um esforço inicial na área de monitoramento de sistemas em tempo real para o grupo do Laboratório de Engenharia de Software da Universidade de Brasília (LES-UnB). Portanto, apesar de já mostrar sinais de que essa solução é possível, ainda há um caminho a se percorrer no sentido de se encontrar soluções que sejam mais viáveis, especialmente em termos de algoritmos específicos para alcançar tal feito.

Dessarte, encerra-se este trabalho ressaltando a ideia de que é possível utilizar *design* baseado em modelos (ou *model-based*) e propriedades projetadas em tempo de *design* para monitorar sistemas durante seu tempo de execução. Também é recomendável a pesquisa e utilização de possíveis heurísticas e métodos mais eficazes para realizar esse monitoramento, dada sua possível alta complexidade computacional.

## 5.2 Limitações da solução proposta

Como mencionado na seção de Metodologia, o artefato proposto para a solução possui algumas limitações. Essas limitações são relacionadas tanto a questões de *framework* (no caso, o ROS), como também de algumas questões mais específicas da programação em si do artefato e de como esse se relaciona com o que já estava disponível do projeto da BSN.

Dentre o primeiro tipo mencionado, destaca-se a limitação de que o ROS, por se tratar de um *framework* que utiliza métodos *callbacks* de uma forma bem específica, acaba por vezes não seguindo o fluxo esperado da solução, o que ocorre, por exemplo, nos casos em que recebe mensagens do *bodyhub* e altera as *flags* de execução de uma forma que não se torna fácil de seguir a linha em que está executando, o que em si só acaba por dificultar o entendimento, e necessita de algumas alterações específicas, como foi explicado na seção mencionada.

Além disso, foi possível notar que existem alguns comportamentos transientes que acabaram dificultando na hora de planejar uma solução, como, por exemplo, o fato de que os sensores não enviavam suas mensagens iniciais informando que haviam sido ligados, para os nós monitores, resultando na necessidade da inscrição em um tópico específico (que já era utilizado anteriormente durante a execução normal), e sua eventual desinscrição, para que não houvesse a sobreposição de tópicos num mesmo momento.

O segundo tipo de limitação, por sua vez, é mais relacionada à questão da forma que foi escolhida para a implementação desta solução, pois buscou utilizar-se de apenas um algoritmo que descreve a linha de execução do programa (baseado no autômato), deixando a responsabilidade de qual propriedade seria executada, ou qual nó seria monitorado para arquivos de configuração, o que, apesar de ter tornado a solução mais dinâmica em relação

a algumas coisas, tornou o funcionamento principal do código mais difícil para futuras alterações.

Por último, vale destacar que, por algum motivo, os nós da BSN da forma que estavam implementados anteriormente ao início deste projeto se encontravam com a necessidade de serem executados em determinada ordem, com o risco de não serem executados de forma alguma no caso dessa ordem não ser obedecida. Isso se deveu, provavelmente, por outros componentes relacionados à arquitetura de auto-adaptação que realizam alguma verificação de ordem de instanciação de nós dentro do ambiente.

Por esse motivo, existe também a limitação sobre quais sensores poderiam vir a ser utilizados como base para a realização deste trabalho, sendo escolhido, portanto, os primeiros a serem iniciados durante a execução regular do sistema.

## 5.3 Trabalhos futuros

Em relação aos trabalhos futuros, existe uma gama de possibilidades que podem ser tomadas. Como mencionado neste capítulo, este trabalho consistiu em um dos primeiros esforços para realizar o monitoramento de sistemas em tempo real no projeto da BSN.

Isso significa que, por mais que o artefato proposto esteja funcionando, não foi planejado levando em consideração alguns aspectos mais desejáveis da engenharia de software, como é possível notar, por exemplo, em seu código, que no momento não é facilmente escalável, o que acaba tornando a solução difícil de ser generalizada para o monitoramento de outras propriedades.

Um trabalho consistindo na refatoração do código do artefato, buscando uma possível forma de torná-lo facilmente extensível e adaptável para a criação mais simples de monitores para outras propriedades seria um trabalho muito interessante de ser realizado.

Assim, partindo do resultado deste trabalho, ou seja, da premissa de que é possível de monitorar propriedades em tempo real para o caso da BSN, pode-se seguir a linha de modelar mais propriedades, inclusive outras presentes em [10], que serviu de referência para as propriedades que foram monitoradas neste trabalho.

Por fim, um outro possível direcionamento é a de otimizar os processos de monitoramento das propriedades, especialmente por meio de heurísticas para sistemas dessa natureza.

# Referências


[1] Bernd Gil, Eric, 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*. arXiv e-prints, página arXiv:2103.14948, março 2021. ix, 4

[2] Rodrigues, Arthur, Genaína Nunes Rodrigues, Alessia Knauss, Raian Ali e Hugo Andrade: *Enhancing context specifications for dependable adaptive systems: A data mining approach*. Information and Software Technology, 112:115–131, 2019, ISSN 0950-5849. `https://www.sciencedirect.com/science/article/pii/S0950584919300941`. ix, 6

[3] Weyns, Danny: *An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective*. John Wiley & Sons, USA, 2021. 1

[4] Knight, J.C.: *Safety critical systems: challenges and directions*. páginas 547–550, 2003. 1

[5] Bengtsson, Johan, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson e Wang Yi: *UPPAAL - a tool suite for automatic verification of real-time systems*. Em *Hybrid Systems*, volume 1066 de *Lecture Notes in Computer Science*, páginas 232–243. Springer, 1995. 2, 10

[6] David, Alexandre, Kim G. Larsen, Axel Legay, Marius Mikucionis e Danny Bøgsted Poulsen: *Uppaal SMC tutorial*. Int. J. Softw. Tools Technol. Transf., 17(4):397–415, 2015. 2, 10

[7] Gil, Eric Bernd, Ricardo Diniz 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*. Em *SEAMS@ICSE*, páginas 224–230. IEEE, 2021. 2

[8] CALDAS, Ricardo Diniz: *An architecture to support control theoretical-based verification of goal-oriented adaptation engines*. Tese de Mestrado, Dissertação (Mestrado em Informática) - Universidade de Brasília, 2019. 2, 7, 17

[9] Caldas, Ricardo Diniz, Arthur Rodrigues, Eric Bernd Gil, Genaína Nunes Rodrigues, Thomas Vogel e Patrizio Pelliccione: *A hybrid approach combining control theory and ai for engineering self-adaptive systems*. Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Jun 2020. `http://dx.doi.org/10.1145/3387939.3391595`. 2

[10] Rodrigues, Arthur, Ricardo Diniz Caldas, Genaína Nunes Rodrigues, Thomas Vogel e Patrizio Pelliccione: *A learning approach to enhance assurances for real-time self-adaptive systems*. Em *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, páginas 206–216, 2018. 2, 7, 38

[11] Quigley, Morgan, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler e Andrew Ng: *ROS: an open-source Robot Operating System*. Volume 3, janeiro 2009. 2, 7

[12] Siegmund, J.: *Program comprehension: Past, present, and future*. Em *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, páginas 13–20, 2016. 8, 9

[13] Soloway, Elliot e Kate Ehrlich: *Empirical Studies of Programming Knowledge*. IEEE Transactions on Software Engineering, SE-10(5):595–609, 1984. 9

[14] Clarke, Edmund M. e E. Allen Emerson: *Design and synthesis of synchronization skeletons using branching time temporal logic*. Em Kozen, Dexter (editor): *Logics of Programs*, páginas 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg, ISBN 978-3-540-39047-3. 10