Estudo e definição do teste de integração de
software para o contexto de sistemas robóticos
móveis

Maria Adelina Silva Brito

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Maria Adelina Silva Brito

Estudo e definição do teste de integração de software para o contexto de sistemas robóticos móveis

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientadora: Profa. Dra. Simone do Rocio Senger de Souza

USP – São Carlos
Junho de 2017

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

B862c Brito, Maria Adelina Silva
Estudo e definição do teste de integração de
software para o contexto de sistemas robóticos móveis
/ Maria Adelina Silva Brito; orientadora Simone do
Rocio Senger de Souza. - São Carlos - SP, 2017.
149 p.

Tese (Doutorado - Programa de Pós-Graduação em
Ciências de Computação e Matemática Computacional)
- Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo, 2017.

1. Teste de software. 2. Teste de integração.
3. Robótica móvel. 4. Cenários de teste. 5. Critérios
de teste. I. Souza, Simone do Rocio Senger de,
orient. II. Título.

Maria Adelina Silva Brito

**Study and definition of the software integration test for
robotic mobile systems**

Doctoral dissertation submitted to the Instituto de
Ciências Matemáticas e de Computação – ICMC-
USP, in partial fulfillment of the requirements for the
degree of the Doctorate Program in Computer Science
and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and
Computational Mathematics

Advisor: Profa. Dra. Simone do Rocio Senger
de Souza

USP – São Carlos
June 2017

*“Este trabalho é dedicado ao meu irmão, Vagno Silva Brito,
que um dia me incentivou a sonhar.”*

AGRADECIMENTOS

A Deus por todos os momentos de conforto e esperança e por proteger e guiar meus caminhos.

A Erinaldo, meu esposo, por entender os momentos ausentes, pela paciência, força e coragem nas horas mais difíceis.

Aos meus pais, Fidelino e Maria do Alívio, muito amados e presentes, sem vocês eu não seria o que sou hoje.

Aos meus irmãos, Wagner, Flávia e Wailson, pela força e por sempre acreditarem em mim. À minha cunhada Poli pelos incentivos. De modo especial, agradeço ao meu irmão Wagner, por um dia ter me motivado fortemente a persistir em busca dos meus sonhos, sendo parte fundamental para que toda esta trajetória fosse possível.

Agradeço também à minha sogra, D. Jurânia, cunhados e sobrinhos, pela motivação e força ao longo do caminho.

À querida orientadora, Simone do Rocio Senger de Souza, pela confiança, incentivo, orientação, paciência e por ter confiado a mim o desenvolvimento desta tese.

Ao professor Paulo Sérgio, pela paciência e direcionamentos durante nossas reuniões e que muito contribuíram para a conclusão deste trabalho.

Aos colegas do LabES e LRM, professores e funcionários.

Ao CNPq e FAPESP pelo apoio financeiro.

*“É preciso força pra sonhar
e perceber que a estrada vai além do que se vê.”
(Los Hermanos)*

RESUMO

BRITO, M. A. S. **Estudo e definição do teste de integração de software para o contexto de sistemas robóticos móveis**. 2017. 149 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2017.

Durante o processo de desenvolvimento de software a atividade de teste pode contribuir com a melhoria de qualidade, produtividade e redução de custos, por meio da aplicação de técnicas e critérios de teste. Esses instrumentos viabilizam a detecção de defeitos, minimizando a quantidade de falhas no produto final. Durante o desenvolvimento de sistemas robóticos móveis, técnicas VST (*Virtual Simulation Tools/Technology*) são enormemente utilizadas. Apesar dos testes usando simulação serem eficientes em alguns casos, eles não permitem uma sistematização da atividade de teste com relação à cobertura de código. Este projeto de doutorado contribui nessa direção, explorando essa lacuna e propondo uma abordagem de teste de integração híbrida, composta por teste funcional e estrutural, a qual explora defeitos relacionados à comunicação entre os processos do sistema robótico. Este estudo foi conduzido com base em conceitos explorados no teste de programas concorrentes, dado que os sistemas robóticos podem apresentar problemas similares aos encontrados nos programas concorrentes, como não determinismo, diferentes possibilidades de comunicação, entre outros. Durante o trabalho foi definida uma abordagem de teste de integração e critérios de cobertura aplicados a sistemas robóticos. Os estudos exploratórios conduzidos indicaram que a abordagem proposta é aplicável e pode contribuir para revelar defeitos não explorados pela simulação e técnicas de teste utilizadas anteriormente.

Palavras-chave: Teste de software, Teste de integração, Robótica móvel, Cenários de teste, Critérios de teste.

ABSTRACT

BRITO, M. A. S. **Study and definition of the software integration test for robotic mobile systems**. 2017. 149 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2017.

During the software development process the test activity contributes in terms of increased software quality, team productivity and cost reduction by application of techniques and test criteria. These mechanisms allow the defects detection, reducing the failures in the final product. In a robotic systems development context, VST techniques (Virtual Simulation Tools/Technology) are widely used. Although these approaches be efficient, in some cases they not allow great systematization of the test activity including code coverage analysis. This project explores this gap and proposes an approach for integration testing applied to robotic embedded systems, composed of functional and structural testing, looking for defects mainly related to the communication among processes of these systems. This study was initially conducted by mapping coverage criteria, defined for concurrent programs testing and exploring problems related to non-determinism, communication, and other undesired situations. During the project we defined an approach of integration testing and coverage criteria for mobile robotic systems. The exploratory case studies indicate that the proposed approach is applicable and can contribute to reveal defects not found using simulations or traditional testing of mobile robotic systems.

Keywords: Software testing, Integration testing, Mobile robotic, Scenarios testing, Criteria testing.

LISTA DE ILUSTRAÇÕES

Figura 1 – Robô Pioneer 3DX com sensores de sonar e laser rangefinder	49
Figura 2 – Grafo de comunicação para um carro robótico	53
Figura 3 – Número de estudos publicados por ano	59
Figura 4 – Estudos por fonte de publicação	59
Figura 5 – Colaborações entre os grupos de pesquisa	60
Figura 6 – Gráfico de bolhas para os estudos selecionados no mapeamento	61
Figura 7 – Automatização das abordagens propostas nos estudos	62
Figura 8 – Taxonomia de falhas para veículos terrestres não tripulados	81
Figura 9 – Diagrama para o robô aspirador de pó	90
Figura 10 – Grafo de comunicação para o sistema do robô aspirador	92
Figura 11 – <i>PSCFG</i> para o sistema do robô aspirador de pó	94
Figura 12 – Carina Parking em ambiente de simulação	101
Figura 13 – Grafo de comunicação para o Carina Parking	101
Figura 14 – <i>PSCFG</i> para o Carina Parking	106
Figura 15 – Execução para o cenário 10, caso de teste 25	110
Figura 16 – Execução para o cenário 3, caso de teste 3	110
Figura 17 – Execução para o cenário 10, caso de teste 24	111
Figura 18 – RobAFIS em ambiente de simulação	112
Figura 19 – Parte do grafo de comunicação para o RobAFIS	113
Figura 20 – <i>PSCFG</i> para o RobAFIS	118

LISTA DE QUADROS

Quadro 1 – Resumo dos conceitos	88
Quadro 2 – Cenários de teste para o sistema	91
Quadro 3 – Resultados das execuções dos programas com defeitos	97
Quadro 4 – Cenários para o Carina Parking	102
Quadro 5 – Cenários para o RobAFIS	115
Quadro 6 – Comparação entre as abordagens de teste aplicadas ao ROS	124

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Código fonte dos processos Collision Avoidance (processo 2) e Controller (processo 4)	145
--	-----

LISTA DE TABELAS

Tabela 1 – Fontes eletrônicas utilizadas nas buscas.	57
Tabela 2 – Estudos selecionados nos dois mapeamentos.	58
Tabela 3 – Estudos selecionados.	62
Tabela 4 – Estudos por técnica de teste.	63
Tabela 5 – Classes de equivalência para os cenários	92
Tabela 6 – Conjunto de casos de teste para o sistema do robô aspirador de pó	93
Tabela 7 – Informações geradas pelo modelo de teste para o programa aspirador de pó	95
Tabela 8 – Elementos requeridos pelos critérios para o programa aspirador de pó	95
Tabela 9 – Classes de equivalência para os cenários do Carina Parking	103
Tabela 10 – Conjunto de casos de teste para o Carina Parking	104
Tabela 11 – Informações geradas pelo modelo de teste para o Carina Parking	105
Tabela 12 – Elementos requeridos para o Carina Parking	107
Tabela 13 – Coberturas para os elementos requeridos para o Carina Parking	108
Tabela 14 – Classes de equivalência para o RobAFIS	116
Tabela 15 – Conjunto de casos de teste para o RobAFIS	117
Tabela 16 – Informações geradas pelo modelo de teste para o RobAFIS	119
Tabela 17 – Elementos requeridos para o RobAFIS	120
Tabela 18 – Coberturas para os elementos requeridos para o RobAFIS	121

LISTA DE ABREVIATURAS E SIGLAS

AADL	Architecture Analysis and Design Language
API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
GFC	Grafo de Fluxo de Controle
GPS	Global Positioning System
GSC	Grafo de Sequência de Comunicação
IDL	Interface Description Language
IEC	International Electrotechnical Commission
IP	Internet Protocol
ISO	International Organization for Standardization
MEF	Máquina de Estados Finita
MI	Mutação de Interface
MPI	Message Passing Interface
PCFG	Parallel Control Flow Graph
PSCFG	Publish/Subscribe-based Control Flow Graph
ROS	Robotic Operational System
RTC	Robot Technology Component
SE	Sistema Embarcado
SO	Sistema Operacional
SUT	System Under Test
TCP	Transmission Control Protocol
UML	Unified Modeling Language
URC	Ubiquitous Robot Companion
VSTs	Virtual Simulation Tools/Technology
VV&T	Verificação, Validação e Teste
XML	eXtensible Markup Language

SUMÁRIO

1	INTRODUÇÃO	29
1.1	Contexto deste projeto	29
1.2	Motivação	31
1.3	Objetivos	32
1.3.1	<i>Objetivo geral</i>	32
1.3.2	<i>Objetivos específicos</i>	32
1.4	Metodologia da pesquisa	32
1.5	Contribuições	33
1.6	Organização da tese	33
2	FUNDAMENTOS SOBRE TESTE DE SOFTWARE	35
2.1	Considerações iniciais	35
2.2	Conceitos básicos sobre teste de <i>software</i>	35
2.3	Teste de integração	39
2.4	Teste de programas concorrentes	44
2.5	Considerações finais	46
3	SISTEMAS ROBÓTICOS MÓVEIS	47
3.1	Considerações iniciais	47
3.2	Robôs móveis	47
3.3	Desenvolvimento de sistemas robóticos móveis	48
3.3.1	<i>Exemplo de um modelo gerado com o ROS</i>	52
3.4	Considerações finais	53
4	ESTADO DA ARTE SOBRE TESTE DE SOFTWARE PARA SIS- TEMAS ROBÓTICOS MÓVEIS	55
4.1	Considerações iniciais	55
4.2	Mapeamento Sistemático sobre teste de integração para sistemas robóticos	56
4.3	Análise dos resultados	58
4.3.1	<i>Teste de alcançabilidade</i>	63
4.3.2	<i>Teste de conformidade</i>	64
4.3.3	<i>Teste de componentes</i>	65
4.3.4	<i>Teste de cenários</i>	70

4.3.5	<i>Teste de desempenho</i>	71
4.3.6	<i>Teste de rede</i>	71
4.3.7	<i>Teste de robustez</i>	72
4.3.8	<i>Search-based software testing</i>	73
4.3.9	<i>Teste de sistema</i>	74
4.3.10	<i>Testbed</i>	74
4.4	<i>Considerações finais</i>	75
5	TESTE DE INTEGRAÇÃO BASEADO NO MODELO DE COMUNICAÇÃO PUBLISH/SUBSCRIBE	77
5.1	<i>Considerações iniciais</i>	77
5.2	<i>Comparação entre programas concorrentes e o modelo de comunicação Publish/Subscribe</i>	78
5.3	<i>Defeitos comuns em robôs móveis</i>	81
5.4	<i>Fundamentos do teste baseado no modelo de comunicação Publish/Subscribe</i>	83
5.5	<i>Abordagem de teste para comunicação Publish/Subscribe</i>	85
5.5.1	<i>Teste funcional: cenários de teste</i>	88
5.5.2	<i>Teste estrutural: critérios de teste</i>	89
5.5.2.1	<i>Exemplo de aplicação</i>	90
5.6	<i>Considerações finais</i>	98
6	AVALIAÇÃO DA ABORDAGEM DE TESTE DE INTEGRAÇÃO PARA SISTEMAS ROBÓTICOS MÓVEIS	99
6.1	<i>Considerações iniciais</i>	99
6.2	<i>Estudo de caso exploratório I: Carina Parking</i>	100
6.2.1	<i>Execução do teste - Carina Parking</i>	102
6.2.2	<i>Discussão sobre o estudo de caso exploratório I</i>	108
6.3	<i>Estudo de caso exploratório II: RobAFISApplication</i>	112
6.3.1	<i>Execução do teste - RobAFISApplication</i>	114
6.3.2	<i>Discussão sobre o estudo de caso exploratório II</i>	121
6.4	<i>Relação critério tipo de defeito</i>	122
6.5	<i>Comparação com outras abordagens de teste para robôs móveis</i>	123
6.6	<i>Considerações finais</i>	125
7	CONCLUSÃO	127
7.1	<i>Contribuições e limitações da pesquisa</i>	128
7.2	<i>Lições aprendidas</i>	129
7.3	<i>Oportunidades para trabalhos futuros</i>	130

REFERÊNCIAS 131

APÊNDICE A CÓDIGO FONTE SISTEMA DO ROBÔ ASPIRADOR

DE PÓ 145

INTRODUÇÃO

1.1 Contexto deste projeto

Teste de *software* é uma das atividades de Verificação, Validação e Teste (VV&T) utilizada para aumentar a qualidade do *software* ou modelo que está sendo desenvolvido. Ele é uma atividade dinâmica, cujo objetivo é revelar a presença de defeitos, caso existam (MYERS, 1979). Além disso, o teste visa não somente revelar a presença de defeitos mas também mostrar que o sistema está de acordo com a especificação.

Técnicas e critérios de teste foram desenvolvidos para maior sistematização da atividade, considerando duas questões: i) seleção dos casos de teste e; ii) adequação dos casos de teste em relação ao programa em teste (HOWDEN, 1975; LASKI; KOREL, 1983; RAPP; WEYUKER, 1985; COWARD, 1988; URAL; YANG, 1988; MALDONADO, 1991; MYERS *et al.*, 2004). Para isso, a atividade de teste inclui basicamente três fases: teste de unidade, teste de integração e teste de sistema. Para o teste de unidade, os casos de teste devem exercitar a estrutura interna das menores unidades que compõem o programa. No teste de integração são exercitadas as interfaces e comunicação entre as unidades. No teste de sistema são verificados os requisitos funcionais e a comunicação do sistema ou modelo em teste com outros sistemas, modelos ou *hardware*.

A robótica é a ciência que interage com o mundo físico por meio de dispositivos computacionais (THRUN; BURGARD; FOX, 2005). Nos sistemas robóticos, a percepção do ambiente externo é realizada por meio de sensores, como câmeras, radares, Global Positioning System (GPS), laserres (*laser range finders*), unidades de medida inercial, dentre outros. A interação com o mundo físico é possível por meio de atuadores como rodas, garras e outros. Robôs móveis são uma classe de robôs que possuem uma plataforma capaz de mobilidade no ambiente (ar, terra ou água) e que serve como base para qualquer tarefa a ser executada. Dessa forma, o princípio básico de um sistema robótico móvel é que ele possibilite ao robô navegar de uma posição conhecida para uma nova posição, evitando obstáculos. Eles são normalmente sistemas

distribuídos, pois são formados por componentes heterogêneos, computadores e dispositivos que interagem para realização das tarefas.

O modelo de comunicação *publish/subscribe* é bastante utilizado nesses sistemas, por possibilitar fraco acoplamento em configurações de grande escala, isto é, a aplicação que envia ou a que recebe uma mensagem não necessita de informações sobre a outra que envia ou recebe a mensagem. Esse modelo de comunicação representa o sistema como um conjunto de nós e arestas. Os nós podem ser produtores ou assinantes de conteúdo. Nessa representação, nós são processos que realizam computação. Os nós assinantes ou *subscribers* expressam interesse em um evento ou padrão de eventos e são notificados sobre eventos gerados por um nó produtor ou *publisher* que tenha seu interesse registrado (EUGSTER *et al.*, 2003). Múltiplos produtores e assinantes podem ser conectados em um simples tópico e um nó pode publicar e/ou subscrever para múltiplos tópicos. Estas características comuns, isto é, correlação de eventos, canais de comunicação, aspectos de *timestamps*, etc., requerem abordagens de teste específicas, que podem mostrar defeitos em aplicações críticas desse domínio.

A atividade de teste em sistemas robóticos, muitas vezes é conduzida utilizando teste funcional e/ou simulações. Apesar de serem sistemas que compreendem programas escritos em C, C++, Python ou outras linguagens, a atividade de teste nesses sistemas é um desafio, pois, características como não-determinismo das entradas, fraca interação com um usuário humano e comunicação entre os processos executando em paralelo, devem ser consideradas.

Kang *et al.* (2010) propõem uma ferramenta de teste de interface (SITAT) para componentes robóticos em ambiente de simulação. Uma abordagem similar é apresentada em Kang e Park (2011), que propõem um método de teste baseado em especificação de interfaces requeridas e fornecidas (RPIST). Ambas abordagens requerem a especificação das interfaces da aplicação. Fan *et al.* (2009) definiram um modelo de teste hierárquico e um *framework* de teste para componentes de tecnologia robótica (RTC). Um *framework* de teste de unidade e integração built-in chamado ROSTest foi definido para apoiar teste de *software* de sistemas desenvolvidos em ROS (ARAÚJO, 2012). ROSTest é compatível com os *frameworks* xUnit, permitindo realizar teste de integração para múltiplos nós. As principais desvantagens do *framework* são a necessidade da escrita de código de teste e de alteração desses códigos, caso o código-fonte da aplicação sofra alterações. Virtual Simulation Tools/Technology (VSTs) têm sido muito usadas em ambientes de desenvolvimento para modelar, simular e avaliar diferentes aspectos de um sistema antes da implementação na plataforma objetivo (FERNANDES *et al.*, 2014). Essas ferramentas reduzem o tempo de desenvolvimento, pois possibilitam que todas as tarefas sejam realizadas no mesmo ambiente.

A validação de sistemas robóticos inclui desafios em razão da quantidade de recursos, equipamentos e suporte técnico necessário para a produção de resultados confiáveis. Determinadas tarefas podem incorrer em altos riscos e incertezas, especialmente aquelas que envolvem transferência de resultados de simulações para o mundo real (CHEN *et al.*, 2009). Abordagens

atuais de teste de integração para sistemas robóticos focam na comunicação entre componentes de *hardware* e *software*, interfaces da aplicação ou requisitos funcionais. Uma perspectiva a ser explorada no teste de robôs móveis é considerar questões que vêm sendo investigadas no contexto de teste de aplicações distribuídas, especificamente aquelas que exploram comunicação e sincronização de tarefas. Não foram encontradas na literatura abordagens de teste de *software* para sistemas robóticos, explorando por meio de critérios de cobertura, características inerentes aos programas distribuídos como concorrência e não determinismo. Critérios de teste de cobertura podem contribuir com este cenário adicionando mecanismos para medir a qualidade durante o desenvolvimento e validação dos sistemas.

A proposta definida durante este doutorado consiste de uma abordagem de teste de integração para sistemas robóticos, em especial, sistemas robóticos compostos de um conjunto de processos distribuídos (tais como sensores e atuadores) que comunicam-se por passagem de mensagem usando o modelo *publish/subscribe*. A atividade de teste de integração é uma emergente e promissora área de pesquisa, mas ainda carece de critérios de teste para revelar falhas no domínio de sistemas robóticos.

1.2 Motivação

Testar sistemas robóticos com eficiência tem se tornado um importante fator para o sucesso das aplicações desenvolvidas nesse domínio. Além do apelo mercadológico, existe a questão da criticidade dessas aplicações, que exige testes mais robustos e com maiores garantias de que a aplicação possui um número de erros reduzido. Durante o desenvolvimento de sistemas robóticos, normalmente, são utilizados *frameworks* e ambientes nos quais as simulações e os testes são conduzidos, antes que o *software* seja embarcado no *hardware*. O problema dessa abordagem é que esses ambientes não fornecem suporte à simulação de todos os possíveis defeitos, principalmente os relacionados a dados não esperados ou problemas nas trocas de mensagens entre os processos. Além disso, os testes normalmente não exploram características de integração entre os processos da aplicação.

Portanto, com base no problema caracterizado a tese aqui apresentada pode ser enunciada da seguinte forma: apoiar a condução da atividade de teste em sistemas robóticos com uma abordagem de teste de integração, que explore a comunicação entre os processos de um sistema distribuído, aplicável e capaz de auxiliar na sistematização da atividade neste domínio.

Nos sistemas robóticos a comunicação entre os componentes e os problemas decorrentes dela assemelham-se à troca de mensagens existente nos programas concorrentes. Considerando essas características a exploração de questões que vêm sendo tratadas no âmbito de programas concorrentes (SOUZA *et al.*, 2008; SARMANHO *et al.*, 2008; SOUZA *et al.*, 2011) pode fornecer benefícios à atividade de teste nesse contexto.

Este trabalho está relacionado ao Instituto Nacional de Ciência e Tecnologia em Sistemas

Embarcados Críticos (INCT/SEC). Esse instituto é coordenado pelo Prof. Dr. José Carlos Maldonado, com financiamento do CNPq e da FAPESP e constitui-se como uma rede de pesquisa apoiada pelo Ministério de Ciência e Tecnologia, cuja missão é agregar habilidades, competências e infra-estrutura necessárias para o desenvolvimento de sistemas embarcados críticos, constituindo-se em aplicações em segurança e agricultura. A abordagem de teste proposta nesta tese apoia o desenvolvimento de aplicações robóticas móveis, desenvolvidas nesse contexto contribuindo com a melhoria da qualidade desses sistemas.

1.3 Objetivos

1.3.1 *Objetivo geral*

Desenvolver uma abordagem de teste de integração para sistemas robóticos móveis.

1.3.2 *Objetivos específicos*

- Investigar os tipos de integração em sistemas robóticos móveis e os problemas relacionados a eles, como, por exemplo, problemas de comunicação e sincronização;
- Estudar as similaridades dos conceitos de programas concorrentes e comunicação em sistemas robóticos móveis;
- Investigar mecanismos de apoio à aplicação da abordagem de teste, incluindo a geração de um modelo e critérios de teste;
- Conduzir estudos experimentais para avaliação da abordagem proposta.

1.4 Metodologia da pesquisa

As características de comunicação dos programas concorrentes assemelham-se aos comportamentos de sistemas robóticos móveis, nos quais vários componentes interagem para realizar uma computação. Algumas abordagens de teste de integração em programas concorrentes exploram comunicação e sincronização entre processos (BEN-ASHER *et al.*, 2006; SOUZA *et al.*, 2008; SOUZA *et al.*, 2011; CHEN *et al.*, 2012). Essas abordagens foram revisitadas, verificando similaridades com o teste de sistemas robóticos móveis que usam o modelo *publish/subscribe* e, assim, a abordagem aqui apresentada é inspirada nos critérios de teste para programas concorrentes (SOUZA; SOUZA; ZALUSKA, 2014).

Após esse estudo comparativo, foi realizada uma revisão de literatura, por meio de um mapeamento sistemático, em que o objetivo foi identificar estudos primários sobre o teste de sistemas embarcados e robóticos que contribuiriam com a proposição da abordagem de teste.

Foram conduzidos estudos utilizando exemplos e sistemas robóticos, com o objetivo de conhecer a estrutura e o comportamento desses sistemas, atividade necessária para a sistematização da abordagem proposta.

Após a condução dos estudos, a abordagem definida nesta tese utiliza teste funcional e estrutural para explorar, principalmente, a comunicação entre os processos de um sistema robótico que utilizam o padrão *publish/subscribe*, implementado no *middleware* ROS. Dois estudos de caso exploratórios foram conduzidos com o objetivo de avaliar a aplicabilidade da abordagem.

1.5 Contribuições

As principais contribuições desta tese para o teste de integração de sistemas robóticos móveis são:

- Estudo sistemático sobre relacionamentos entre o teste de programas concorrentes e o teste de sistemas robóticos móveis;
- Desenvolvimento de uma abordagem de teste de integração para sistemas robóticos móveis;
- Definição de um modelo e critérios de teste de integração de cobertura para extrair requisitos de teste;

1.6 Organização da tese

A tese é composta por 7 capítulos, que estão assim estruturados:

- O Capítulo 2 descreve os trabalhos relacionados ao teste de *software*.
- O Capítulo 3 aborda as principais características dos robôs móveis e seu desenvolvimento.
- O Capítulo 4 apresenta as abordagens que vêm sendo utilizadas no teste de sistemas robóticos.
- O Capítulo 5 reporta a abordagem de teste de integração desenvolvida durante este doutorado.
- O Capítulo 6 descreve os estudos de caso exploratórios conduzidos para validação da abordagem proposta.
- O Capítulo 7 apresenta as contribuições, limitações e oportunidades para trabalhos futuros a partir deste trabalho de doutorado.

FUNDAMENTOS SOBRE TESTE DE SOFTWARE

2.1 Considerações iniciais

Neste capítulo são apresentados conceitos de teste de *software* com ênfase no teste de integração. Também são reportadas outras abordagens de teste, que exercitam trocas de mensagens entre componentes ou processos e apresentam similaridades com o teste de integração, como o teste de programas concorrentes.

O capítulo está organizado da seguinte forma: na Seção 2.2 são apresentados os principais conceitos de teste de *software*, incluindo as fases e técnicas de teste. Na Seção 2.3 estão descritos mais detalhes sobre o teste de integração, em razão do objetivo desta proposta de doutorado explorar esse tipo de teste. A Seção 2.4 apresenta conceitos sobre o teste de programas concorrentes. E por fim, na Seção 2.5 estão as considerações finais deste capítulo.

2.2 Conceitos básicos sobre teste de *software*

Alguns conceitos relacionados ao teste de *software* costumam gerar dúvidas, tais como falha, defeito, erro e engano. Neste trabalho são adotadas as seguintes definições (DELAMARO; MALDONADO; JINO, 2007): defeito (*fault*) consiste em um passo incorreto, processo ou definição de dados incorreta em um programa, engano (*mistake*) é uma ação humana capaz de produzir um defeito, erro (*error*) caracteriza-se por um estado inconsistente durante a execução do programa e falha (*failure*) é um resultado diferente do esperado.

Teste é uma das atividades de VV&T dinâmicas mais utilizadas. Adotando uma definição mais simplista, o teste consiste em executar um programa com o objetivo de encontrar defeitos (MYERS *et al.*, 2004). Dessa forma, testar um programa ou modelo é executá-lo inserindo dados de teste (entradas) e comparando as saídas obtidas com as previstas na especificação. Caso

seja obtida uma saída não correspondente a esperada, o teste revelou uma falha no programa ou modelo, sendo necessário uma revisão com o objetivo de identificar onde o defeito se encontra, atividade conhecida como depuração.

Para que os testes sejam realizados durante todo o processo de desenvolvimento estão previstas três fases: teste de unidade, teste de integração e teste de sistema. O objetivo do teste de unidade é testar as menores partes que compõem o sistema, como métodos, funções ou classes, de maneira independente. Nesse teste, busca-se encontrar defeitos de programação, algoritmos incorretos ou mal implementados, estruturas de dados incorretas, ou outros tipos de defeitos, limitando-se o teste a lógica interna e estruturas de dados dentro dos limites da unidade. O teste de integração verifica as estruturas de comunicação entre as unidades que compõem o sistema. Técnicas que exercitam caminhos específicos do programa são usadas para garantir a cobertura dos principais caminhos de controle (PRESSMAN, 2005). Para o projeto de casos de teste são mais utilizadas técnicas que exploram entradas e saídas. No teste de sistemas o objetivo é verificar se as funcionalidades foram implementadas corretamente, se os requisitos não funcionais satisfazem a especificação e características como correção, coerência e completude também são avaliadas (DELAMARO; MALDONADO; JINO, 2007). A aplicação é testada com outros sistemas de *software* e elementos de *hardware* que possivelmente estarão utilizando-o após sua liberação.

O ideal na atividade de teste seria a realização de testes exaustivos, isto é, testes que submeteriam o programa a todas as entradas possíveis para verificar se as saídas corresponderiam ao esperado. O problema dessa abordagem é que o domínio de entrada dos programas é muitas vezes grande, ou até infinito e impossibilita testes dessa forma, limitando a execução dos testes a algumas entradas do domínio. Em razão dessa limitação não se pode afirmar que um programa está isento de defeitos. A fim de aumentar a abrangência dos testes no escopo de um sistema foram criadas técnicas e critérios de cobertura. Os critérios de teste são “regras” utilizadas para identificar diferentes subdomínios do conjunto de entrada de um programa. Um subdomínio é parte do domínio de entrada em que os elementos possuem as mesmas características. Para identificar os subdomínios é necessário satisfazer os requisitos de teste, exemplo de um critério de teste seria por exemplo, executar todos os nós de um grafo de fluxo de controle que representa um programa. Os dados de teste que satisfazem um mesmo requisito pertencem ao mesmo subdomínio e os critérios de teste constituem-se em diferentes regras que determinam os subdomínios. As técnicas que determinam os critérios de teste são classificadas de acordo com a origem da informação que é utilizada para estabelecer os requisitos de teste (MALDONADO, 1991). As principais técnicas de teste são:

- **Técnica de teste funcional:** considera o sistema uma “caixa preta” e os dados de entrada são gerados com base na especificação, requisitos e projeto do sistema. Na técnica funcional os detalhes de implementação não são considerados, pois o objetivo é testar as funções do sistema. Como todos os critérios da técnica funcional baseiam-se na especificação, é

fundamental que esta reflita as necessidades do sistema que será projetado e possua menos ambiguidades possíveis para facilitar a geração de casos de teste efetivos em detectar os defeitos que possam existir. Uma grande vantagem dos testes funcionais é que eles podem ser aplicados em todas as fases e em qualquer programa, pois nesse tipo de teste não são avaliados os detalhes de implementação do sistema. Alguns critérios dessa técnica são: Particionamento de Equivalência, Análise do Valor Limite, Grafo Causa-Efeito, *Error-Guessing*, Teste Funcional Sistemático e Método de Partição-Categoria (MYERS *et al.*, 2004; LINKMAN; VINCENZI; MALDONADO, 2003; OSTRAND; BALCER, 1988). Uma limitação dos critérios dessa técnica é a não garantia de que partes críticas do sistema tenham sido testadas.

- **Técnica de teste estrutural:** também conhecida como teste “caixa branca”. Nessa técnica os casos de teste são gerados com base na implementação do *software* e avaliam se a estrutura interna (código-fonte) da unidade está correta com vistas à especificação. A maioria dos critérios dessa técnica utiliza uma representação do programa chamada Grafo de Fluxo de Controle (GFC), composto por nós que representam conjuntos de comandos executados sequencialmente e arestas que representam transferências de controle entre comandos (MYERS, 1979). O Grafo Def-Uso é uma extensão do GFC, em que são acrescentadas informações sobre definição e uso de variáveis nos nós e arcos, representando o fluxo de dados do programa. Uma definição de variável ocorre quando seu valor é armazenado em uma posição de memória. Para isso, a variável deve se encontrar do lado esquerdo de um comando de atribuição ou ser passada por referência em alguma função, armazenando o valor de saída dessa função ou o valor de retorno. Um uso de uma variável ocorre em um comando no qual o valor da variável é acessado. Os casos de teste visam a percorrer os caminhos internos possíveis da unidade, buscando descobrir defeitos no uso de variáveis, laços e condições. Um problema do teste estrutural é a presença de elementos requeridos não executáveis que impossibilitam testar 100% da unidade. Os critérios da técnica estrutural são classificados com base em: i) fluxo de controle: baseiam-se no GFC e utilizam características do código do programa, como comandos ou desvios, para determinar as estruturas que devem ser testadas. Os critérios mais conhecidos dessa classe são: todos-nós, todas-arestas e todos-caminhos (PRESSMAN, 2000; MYERS *et al.*, 2004); ii) complexidade: utilizam informações sobre a complexidade do programa para gerar os requisitos de teste. Como exemplo dessa classe está o critério de McCabe (1976); iii) fluxo de dados: selecionam caminhos de um programa com base nas definições e usos das variáveis (PRESSMAN, 2005). Alguns deles são: critérios de Rapps e Weyuker (1985) e Potenciais-Usos (MALDONADO, 1991).
- **Técnica de teste baseada em defeitos:** são utilizados os defeitos mais comuns que ocorrem durante o processo de desenvolvimento de *software* para derivar os requisitos de teste. O objetivo da técnica é construir casos de teste que revelem a presença ou ausência de

defeitos específicos (DEMILLO, 1986). O teste de mutação ou análise de mutantes é um critério da técnica baseada em defeitos, em que o artefato (programa ou especificação) em teste é alterado com a utilização dos operadores de mutação, como se “defeitos” estivessem sendo inseridos. Os artefatos gerados com os defeitos modelados pelos operadores de mutação são chamados mutantes. Na fase de execução dos testes, casos de teste são gerados para cobrir os subdomínios modelados pelos operadores de mutação. O objetivo é obter saídas distintas para o programa original e para o programa mutante. Caso a saída obtida a partir do programa mutante seja diferente da obtida para o programa original, o mutante deve ser descartado, ou “morto”, pois, o caso de teste foi capaz de expor a diferença entre o programa original e o programa mutante, significando que o defeito modelado pelo mutante não está presente no programa. Um caso de teste é um par formado por entrada e saída esperada. Caso as saídas sejam iguais, duas situações podem ter ocorrido: ou o caso de teste não foi suficiente para identificar a diferença entre os dois programas ou os programas são equivalentes. Os mutantes equivalentes geram as mesmas saídas que o programa original para qualquer valor do domínio de entrada e devem ser identificados e marcados como equivalentes. Um defeito é identificado quando um mutante que não é equivalente ao programa original obtém a saída correta e esperada, nesse caso, o mutante é denominado *error-revealing*.

Além dessas três técnicas há autores que consideram também a técnica de teste baseada em modelos, que refere-se ao ato de modelar e gerar testes a partir de modelos formais de comportamento da aplicação (MATHUR, 2008). Os modelos formais possibilitam resolver imprecisões originadas de linguagens naturais que podem causar problemas, pois as linguagens naturais geralmente são ambíguas e imprecisas por natureza. Para aplicar teste baseado em modelos, inicialmente o sistema deve ser modelado considerando as sequências de ações que podem ocorrer durante seu uso. Como exemplo de um modelo formal, as “Máquinas de Transições de Estados” são técnicas que permitem descrever as próximas ações em um grafo em que os nós representam as configurações ou estados e as arestas representam as transições entre as configurações. Dentre essas técnicas o modelo mais simples é o de Máquina de Estados Finita (MEF). Uma MEF é uma máquina hipotética composta por estados e transições. Cada transição liga um estado a a um estado b , sendo que a e b podem ser o mesmo estado. A cada instante a máquina pode estar em apenas um de seus estados. Em resposta a um evento de entrada, a máquina gera um evento de saída e muda de estado. Uma MEF pode ser representada tanto por um diagrama de transição de estados quanto por uma tabela de transições. Uma sequência de teste é uma sequência de símbolos de entrada derivada da MEF, que quando submetida à implementação verifica a conformidade da saída gerada pela implementação com a saída especificada na sequência de teste. Os métodos clássicos para a geração de sequências de teste a partir de MEFs são: *Transition Tour* (TT), *Unique input/output* (UIO), *sequência de distinção* (DS) e W (SIMÃO, 2007). Outros formalismos para derivação de dados de teste são *statecharts*,

geração por prova de teorema, programação com restrições, execução simbólica, cadeias de Markov, dentre outros. *Model checking* é um método que permite a verificação de uma ou mais propriedades de um modelo de uma aplicação.

2.3 Teste de integração

O teste de integração foca nas interfaces das unidades que compõem o sistema. Dependendo do paradigma de programação utilizado uma unidade pode ser um módulo, uma classe, um componente, etc. Neste trabalho utiliza-se o termo “unidade” em um contexto mais geral, significando a parte de um programa que pode ser testada independentemente do programa completo. Assim, um processo de um sistema distribuído é considerado uma unidade e, dessa forma, os conjuntos de teste devem ser construídos com o objetivo de exercitar as interações entre essas unidades. Em geral, os erros de integração são de três tipos (LEUNG; WHITE, 1990):

- **erros de interpretação:** ocorrem quando uma unidade implementa uma funcionalidade diferente da especificação ou interpretação da mesma. Para esse tipo de erro, supondo que a unidade *A* chame a unidade *B*, ele pode ocorrer de três formas: i) funcionalidade incorreta, quando a funcionalidade implementada na unidade *B* pode não ser a requerida pela especificação de *A*; ii) funcionalidade extra, quando a unidade *B* contém uma ou mais funcionalidades extra não requeridas pela unidade *A* e; iii) funcionalidade ausente, quando *A* passa valores para *B* fora do domínio de *B*.
- **chamadas incorretas:** ocorrem quando há uma chamada a outra unidade no local incorreto. Tal erro pode ocorrer devido a três tipos de defeitos: i) chamada extra, quando a chamada encontra-se em um caminho que não deveria estar; ii) chamada incorretamente posicionada, na qual a chamada ocorre em um caminho que deveria possuí-la mas no local incorreto e; iii) chamada ausente, em que um caminho que deveria ter uma chamada não a possui.
- **erros de interface:** ocorrem quando o padrão de interação entre duas interfaces é violado, como exemplos: ordem incorreta de parâmetros em um método, tipo de parâmetro incorreto, formato incorreto, entre outros.

No teste de unidade as definições e usos de variáveis estão na mesma unidade. Durante o teste de integração isso não ocorre, definições e usos estão em diferentes unidades e são responsáveis pela comunicação entre elas. O teste de integração pode ser realizado em várias situações: o testador pode necessitar testar exatamente a comunicação entre determinadas unidades, testar uma unidade que depende de outra ainda não implementada e/ou analisar a integração de unidades incrementalmente. Para facilitar essa atividade, podem ser utilizadas as seguintes estruturas:

- **driver**: é uma unidade que exercita as interfaces da unidade em teste. O *driver* simula uma unidade que chama a unidade sendo testada, fornecendo uma entrada e controlando a saída para um dado caso de teste;
- **stub**: é um programa simples que possui as mesmas interfaces de uma unidade chamada. Um *stub* substitui unidades ainda não integradas.

Os *drivers* e *stubs* auxiliam na integração das unidades que são testadas conforme o sistema é desenvolvido. Para o teste de integração duas questões são importantes: quais unidades são o foco do teste e em que sequência as interfaces entre as unidades devem ser exercitadas. Essas questões estão sempre presentes porque o *software* pode ser integrado de várias formas dependendo dos objetivos do teste (AMMANN; OFFUTT, 2008).

Seguindo uma estrutura de projeto não incremental pode-se utilizar no teste de integração a estratégia *big bang*, na qual o teste de integração é aplicado com o sistema já integrado. Após o teste individual das unidades, estas são integradas e testadas. As desvantagens desse tipo de teste são a difícil localização de defeitos, unidades críticas tratadas como as demais e utilização de pequenos conjuntos de teste para revelar defeitos nas interfaces.

As principais estratégias incrementais são a *top-down*, a *bottom-up* e a *sandwich*. Na estratégia *top-down* as unidades são adicionadas numa ordem hierárquica, começando do nível mais elevado ou seja pelas unidades com maiores dependências. O teste tem início com a unidade do programa principal e as unidades de níveis mais baixos, chamadas pela unidade principal, são substituídas por *stubs*. À medida que o teste avança, os *stubs* são aos poucos substituídos pelo código e retestado. Como vantagens estão: as unidades são depurados separadamente e o teste das principais funções de controle é realizado no início, fazendo com que problemas sejam revelados mais cedo. Nessa estratégia duas abordagens podem ser utilizadas:

- **em largura**: inicia-se com a integração das unidades com base nas dependências;
- **em profundidade**: integram-se as unidades com base na implementação de uma dada funcionalidade do *software*.

Na estratégia *bottom-up* as unidades são adicionados em etapas seguindo a ordem de dependência, iniciando com as unidades com menores dependências (ordem de baixo para cima). Nesse tipo de estratégia são usados *drivers* para substituir e simular algumas unidades durante o teste. As vantagens dessa estratégia são: unidades depuradas separadamente, teste mais simples das unidades de nível mais alto e projeto de casos de teste facilitado pela ausência de *stubs*. A estratégia *sandwich* combina as estratégias *top-down* e *bottom-up*. Sua aplicabilidade deve ser avaliada caso a caso. Para sua utilização define-se uma linha base na estrutura de integração das unidades. As unidades acima da linha são testadas usando a estratégia *top-down* e as unidades abaixo da linha são testadas usando *bottom-up*.

Pressman (2005) considera que abordagens incrementais são capazes de encontrar defeitos mais facilmente, pois as interfaces são testadas de forma mais completa. Porém, outro estudo realizado por Solheim e Rowland (1993) mostra que não há diferença significativa entre as estratégias com relação à capacidade em revelar defeitos. Diante dessas conclusões para realizar o teste de integração uma boa estratégia pode ser definir inicialmente as unidades críticas e priorizá-las, pois quanto antes estas unidades forem testadas, menores são as chances de localizar defeitos tardiamente. Dependendo da posição da unidade na estrutura do programa escolhe-se a estratégia de teste de integração mais adequada. Uma unidade aqui é considerada crítica se aborda diversos requisitos de *software*, possui elevado nível de controle ou muitas dependências, é complexa e propensa a erros e possui restrições de desempenho definidas.

Alguns trabalhos enfatizam a definição de critérios de teste para testar com maior eficiência a interação entre as unidades de um sistema. Linnenkugel e Müllerburg (1990) definiram critérios baseados no fluxo de controle e no fluxo de dados interprocedimentais. Os critérios baseados no fluxo de controle utilizam um Grafo de Chamadas para representar o programa e gerar os elementos requeridos. Um Grafo de Chamadas é um multigrafo direcionado em que cada nó representa uma unidade e cada aresta representa a chamada de uma unidade a outra. Alguns dos critérios baseados no fluxo de controle são: i) todos-módulos, que requer que cada unidade seja exercitada pelo menos uma vez; ii) todas-relações, que requer que pelo menos uma aresta entre dois módulos seja exercitada pelo menos uma vez; iii) todas-relações-múltiplas, que requer que todas as arestas entre dois módulos sejam executadas pelo menos uma vez. Os critérios baseados no fluxo de dados exploram associações definição-uso interprocedimentais a serem cobertas pelos casos de teste. Como exemplo, dada uma variável de comunicação v de uma função F , alguns dos critérios são: i) INT-todas-defs, que requer para cada chamada a execução de um caminho livre de definição com relação a v , de cada definição de v antes da chamada de F até algum ponto em que v é usada em F e de cada definição de v dentro de F até algum uso após o retorno de F ; ii) INT-todos-usos, que requer para cada chamada a execução de um caminho livre de definição com relação a v de cada definição de v antes da chamada de F até todos os pontos onde v é usada em F e de cada definição de v dentro de F até todos os usos após o retorno de F .

Harrold e Soffa (1991) definiram critérios de fluxo de dados utilizando o Grafo de Fluxo Interprocedimental, que somado aos Grafos de Fluxo de Controle de cada unidade representam o fluxo de dados interprocedimental. Os Grafos de Fluxo Interprocedimentais abstraem informações sobre os fluxos de controle e de dados dos procedimentos. Os nós representam partes do código relacionadas à comunicação entre procedimentos, como pontos de entrada, pontos de saída, chamadas e retornos de procedimentos. As arestas, chamadas arestas de alcance, representam informações sobre o fluxo de controle do procedimento. Além dos critérios de fluxo de dados, é apresentado um algoritmo para avaliar se um determinado caminho executado pelo programa cobre ou não uma associação. Talvez, esta seja a maior contribuição desse trabalho, pois a maior dificuldade no estabelecimento de critérios interprocedimentais está na definição de

meios que possibilitem sua implementação e utilização na prática (DELAMARO, 1997).

Os testes de interfaces ocorrem quando unidades são integradas para criar sistemas maiores. O objetivo é detectar defeitos nas interfaces. O teste de Mutação de Interface (MI) foi inspirado no teste de mutação adicionando características do teste de integração (DELAMARO, 1997). O teste de mutação utiliza os defeitos mais comuns cometidos pelos programadores durante o processo de desenvolvimento para gerar os requisitos de teste. O critério de teste MI permite testar a interação entre as unidades que compõem o *software*, diferindo do teste de mutação, que explora as características das unidades, separadamente. Como apoio ao teste de MI foi desenvolvida a ferramenta de teste *Proteum/AM*, que contém 33 operadores de mutação modelando erros típicos de integração entre as interfaces de um programa escrito em C. Apesar do teste de mutação ser apontado como um dos mais eficientes para testar um determinado programa em nível de unidade, os programas gerados com a inserção dos operadores de mutação são sempre os mesmos. Com o critério MI, que insere defeitos nas conexões entre as unidades, o testador precisa criar casos de teste que revelem as diferenças de comportamento entre o programa original e os mutantes. Dessa forma, se o caso de teste revela um defeito na interface que sofreu mutação, automaticamente foi detectado um defeito de integração entre essas interfaces (VINCENZI *et al.*, 1999).

O paradigma de orientação a objetos surgiu em decorrência das dificuldades na manipulação de programas do paradigma procedimental em que existe uma dependência muito grande entre os procedimentos ou funções e os dados. A programação orientada a objetos melhora esse problema quando agrupa em uma classe os dados ou atributos e os métodos que realizam operações sobre os objetos. Outras características estão presentes em programas Orientados a Objetos (OO) como herança, polimorfismo e acoplamento dinâmico. Com essas características os programas OO apresentam-se mais legíveis, fáceis de compreender, com reúso facilitado e tudo isso contribui para facilitar a manutenção em virtude do ocultamento de informação. Por outro lado, essas mesmas características podem facilitar a introdução de defeitos.

O teste de interfaces é importante também em programas OO, pois os objetos são definidos por suas interfaces (JORGENSEN; ERICKSON, 1994). Há na literatura diferentes conotações para as fases do teste OO como a apresentada por Colanzi (1999): 1) teste de unidade testa os métodos individualmente; 2) teste de classe testa a interação entre métodos de uma classe; 3) teste de integração testa a interação entre classes do sistema e; 4) teste de sistema testa a funcionalidade do sistema como um todo. Outros pesquisadores utilizam outras classificações (ARNOLD; FUSON, 1994; BINDER, 1999; CHEN *et al.*, 1998; PERRY; KAISER, 1990).

Vincenzi *et al.* (2006) estudaram o teste estrutural de unidades para programas OO em Java, em que um método era considerado a menor unidade a ser testada. Nesse trabalho, foi descrita a ferramenta de teste *JaBUTi*, que obtém informações do fluxo de dados sobre as definições e usos de variáveis a partir do *bytecode*. Lemos, Franchin e Masiero (2009) estenderam

o grafo de fluxo para programas orientados a aspectos (Grafo *Aspect-Oriented Def-Use* - AODU) para apoiar o teste de integração par a par entre métodos e adendos de programas OO e AspectJ. Como parte desse trabalho definiu-se o grafo PWDU (*PairWise-Def-Use*) e a ferramenta *JaBUTi* foi estendida para dar apoio ao teste de integração par a par, originando uma versão da ferramenta denominada *JaBUTi/AJ*. Uma proposta para teste estrutural de integração de programas OO e Orientados a Aspectos (OA) é apresentada em [Neves, Lemos e Masiero \(2011\)](#). A abordagem permite que uma unidade, como um método ou adendo, seja testada integrada com todas as unidades com as quais ela interage diretamente pela técnica estrutural usando a ferramenta *JaBUTi/AJ*. [Neves, Delamaro e Masiero \(2014\)](#) definiram uma abordagem de teste para sistemas robóticos utilizando os critérios de teste todos-nós e todas-arestas para verificar a cobertura do código.

Um componente pode ser definido como uma unidade com interfaces especificadas por meio de contratos e com dependências de contexto explícitas ([SZYPERSKI, 2002](#)). Componentes podem ser desenvolvidos por terceiros e reusados em outras aplicações independentemente da linguagem de programação ou ambiente de desenvolvimento, reduzindo tempo e custo de desenvolvimento. Com isso, o desenvolvimento baseado em componentes tem motivado uma reavaliação das estratégias de teste quanto à adequação a esse contexto. O teste baseado em componentes pode ser analisado segundo duas perspectivas ([DELAMARO; MALDONADO; JINO, 2007](#)):

- **do cliente:** que são os desenvolvedores que utilizam em suas aplicações os componentes desenvolvidos por terceiros. Dentre as características que dificultam a atividade de teste nesse contexto estão: 1) nem sempre o cliente tem acesso ao código-fonte do componente, dificultando o uso de técnicas de teste estrutural ou de mutação e; 2) nem sempre o componente e a aplicação usuária do mesmo são escritos na mesma linguagem de programação ([ROSENBLUM, 1997](#)). Por outro lado, os testes funcionais não garantem que partes críticas do sistema são exercitadas, já que os requisitos de teste são baseados na especificação. Em razão da indisponibilidade do código-fonte, critérios alternativos como os baseados em reflexão computacional ([EDWARDS, 2001a](#)), polimorfismo ([TYLER; SOUNDARAJAN, 2004](#)), metadados ([EDWARDS, 2001b](#)), e metaconteúdo (*metacontent*), além dos testes baseados em estados ([BEYDEDA; GRUHN, 2001](#)) e autoteste (*built-in testing*) vêm sendo explorados.
- **do desenvolvedor:** quem implementa e testa o componente sem conhecimento das aplicações que poderão usá-lo. O desenvolvedor acessa o código-fonte, podendo utilizar teste de unidade ou integração tradicional, porém, como ele não possui informações sobre os sistemas que poderão utilizar o componente, os testes realizados pelo desenvolvedor não exercitam situações de possíveis usos futuros do mesmo ([DELAMARO; MALDONADO; JINO, 2007](#)).

Com base nisso, uma estratégia de teste de componentes deve compreender tanto testes unitários, realizados pelo desenvolvedor quanto testes de integração, realizados pelo cliente do componente. No teste de integração o foco principal são as interfaces, pois estas conectam dois ou mais componentes. Interfaces são pontos de interação dos componentes, por meio das quais um componente cliente pode usar os serviços declarados nas interfaces requeridas e/ou fornecidas. Cada interface possui um identificador único e pode incluir várias operações em que cada uma realiza uma determinada tarefa. Os testes de integração de componentes são usualmente na forma de sequências de chamadas de métodos nas interfaces.

Os estudos sobre teste de componentes enfatizam diferentes características, como a geração de dados de teste de integração a partir do teste de unidade (RUBINOV, 2010), diagramas Unified Modeling Language (UML) (HARTMANN; IMOBERDORF; MEISINGER, 2000; LABICHE *et al.*, 2000; MAIBAUM; LI, 2007) ou outros modelos (WU; CHEN; OFFUTT, 2003; YUAN; XIE, 2006). Wu, Chen e Offutt (2003) definem o Grafo de Interação de Componentes (CIG) para modelar as interações e dependências dos componentes de uma aplicação. Essa abordagem é caixa-preta e um CIG é formado por um conjunto de interfaces requeridas e fornecidas e cada vértice representa um método de uma interface. Uma ferramenta para testar programas OO apoiando tanto a geração quando a execução dos testes é apresentada em Maibaum e Li (2007). Nessa abordagem, casos de teste são gerados a partir de diagramas UML e contratos estabelecem relações entre os componentes. Na sequência são gerados códigos a partir dos contratos, que junto aos componentes em teste formam o *framework* de teste.

2.4 Teste de programas concorrentes

Um programa concorrente é composto por vários processos que interagem entre si para resolver um problema. Um processo é um programa em execução, constituído pelo programa executável, seus dados, o contador de instruções, os registradores e todas as informações necessárias a sua execução (TANENBAUM, 2001). A concorrência existe, a partir do momento em que dois ou mais processos iniciaram sua execução, mas não terminaram (ALMASI; GOTTLIEB, 1994). Para existir concorrência é necessário que os processos envolvidos comuniquem-se e se sincronizem. A concorrência pode existir tanto em sistemas com único processador (monoprocessados) como em sistemas multiprocessadores (com mais de um processador). Em sistemas monoprocessados existe a falsa impressão de que os processos estão sendo executados ao mesmo tempo, fato que caracteriza o pseudo-paralelismo. Para que os processos sejam executados em paralelo é necessário que estes sejam executados no mesmo instante de tempo, caracterizando o paralelismo real.

A comunicação permite a um processo influenciar na execução de outros. Para que dois processos se comuniquem, um deve executar uma ação que o outro perceba, como exemplo, alterar o valor de uma variável ou enviar uma mensagem. Já a sincronização pode ser vista

como um conjunto de restrições na ordenação de eventos. O programador pode empregar mecanismos de sincronização para atrasar a execução de um processo obedecendo determinadas restrições (ANDREWS; SCHNEIDER, 1983). A sincronização controla a ordem de execução de cada processo sobre os dados e recursos compartilhados.

Há diferentes conceitos de síncrono e assíncrono e bloqueante e não-bloqueante, a depender do autor ou do contexto. Aqui serão utilizadas as definições apresentadas em Souza, Vergilio e Souza (2007). Na comunicação interprocessos baseada em passagem de mensagens, quando a memória é distribuída, uma mensagem de envio de dados é implementada por um *send* e uma mensagem para que um processo receba dados de outro é implementada com um *receive*. Uma comunicação é considerada síncrona quando o *send* (ou o *receive*) aguarda que o par, nesse caso o *receive* (ou o *send*) seja executado. Para isso, é realizada uma confirmação (*ack*) a cada troca de mensagem. Nesse contexto, toda comunicação síncrona é também bloqueante. Na comunicação assíncrona, o comando *send* (ou *receive*) não necessita esperar o comando *receive* (ou *send*) executar. Um *send* é bloqueante se garante que o *buffer* com a mensagem possa ser reutilizado após a liberação do *send*, sem que a mensagem seja perdida (ou sobreposta). O *receive* será bloqueante até que a mensagem esteja disponível para o processo receptor no *buffer*.

O não determinismo é outra característica dos programas concorrentes, não existente nos programas sequenciais. Nesse caso, se um programa é executado mais de uma vez com a mesma entrada, diferentes sequências de comandos podem ser executadas e nem sempre a saída para uma determinada entrada será a mesma. O resultado da computação depende, além do conjunto de entrada, das sincronizações ocorridas entre os processos da aplicação.

As pesquisas em teste de programas concorrentes tiveram início com estudos teóricos sobre modelos como redes de Petri, grafos de computação e esquemas de programas paralelos para representar o fluxo de controle de processos concorrentes (MILLER, 1973; HERZOG, 1979). Em paralelo, técnicas de teste já utilizadas em programas sequenciais começaram a ser adaptadas para programas concorrentes. O grafo de concorrência foi definido para representar as possíveis sincronizações de um programa concorrente em nível estrutural (TAYLOR, 1983). A partir daí, outros modelos e critérios de teste foram definidos (TAYLOR; LEVINE; KELLY, 1992; YANG; CHUNG, 1992; YANG; SOUTER; POLLOCK, 1998; KOPPOL; TAI, 1996; KOPPOL; CARVER; TAI, 2002; KRAWCZYK; WISZNIEWSKI, 1996; BECHINI; CUTAJAR; PRETE, 1998; PARK *et al.*, 2007). Há vários estudos considerando abordagens de teste de alcançabilidade (LEI; CARVER, 2006; WONG W.E., 2008; CARVER; LEI, 2004; LEI *et al.*, 2007; PU; XU, 2008; YOUNG *et al.*, 1995) e técnicas de teste de mutação ou perturbação (JAGANNATH *et al.*, 2010; SEN; ABADIR, 2010; GLIGORIC; JAGANNATH; MARINOV, 2010).

O grupo de Engenharia de *Software* do ICMC/USP tem desenvolvido modelos, critérios e ferramentas de apoio, considerando ambos os paradigmas de comunicação (SOUZA *et al.*, 2008; SARMANHO *et al.*, 2008; SOUZA *et al.*, 2008; ENDO *et al.*, 2008; HAUSEN *et al.*, 2007; SOUZA *et al.*, 2011; SOUZA; SOUZA; ZALUSKA, 2014; SILVA; SOUZA; SOUZA,

2012; MELO; SOUZA; SOUZA, 2012; BRITO; SOUZA; SOUZA, 2013). Um modelo de teste, denominado Parallel Control Flow Graph (PCFG), foi definido para capturar estaticamente informações sobre o programa concorrente a ser testado, como fluxo de controle e de dados (definição e uso de variáveis) de cada processo, informações sobre mensagens enviadas ou recebidas e todos os possíveis pares de sincronização e comunicação entre os processos. Com base nesse modelo de teste, foi definido um conjunto de critérios que englobam aspectos de fluxo de controle, de dados e de comunicação. Os critérios estão divididos em dois grupos: 1) critérios baseados em fluxo de controle e de comunicação (todos-nós-s, todos-nós-r, todas-arestas-s, todos-nós e todas-arestas) e 2) critérios baseados no fluxo de dados e em troca de mensagens (todas-defs, todos-s-usos, todos-c-usos todos-p-usos, todos-s-c-usos, todos-s-p-usos, dentre outros) (SOUZA *et al.*, 2008). Foram definidas ferramentas para apoiar o teste, destacando-se a ferramenta *ValiMPI*, que apoia o teste de programas concorrentes em Message Passing Interface (MPI) (HAUSEN *et al.*, 2007), a *ValiPthread*, que apoia o teste de programas concorrentes em Pthreads (*multithreaded*) (SARMANHO *et al.*, 2008), a *ValiBPEL-Web*, que apoia o teste de composição de Web Services (ENDO *et al.*, 2008) e a ferramenta *ValiMPI_Mut* que implementa operadores de mutação para programas concorrentes em MPI (SILVA; SOUZA; SOUZA, 2012).

Os critérios de teste (SOUZA *et al.*, 2008) podem ser utilizados para o teste de unidade, no qual cada processo é considerado uma unidade, e para o teste de integração, em que são usados os critérios de sincronização para testar a interação entre os processos. Nesse caso, o teste de integração considera somente interações relacionadas à troca de mensagens ou ao uso de variáveis compartilhadas. Na programação concorrente por troca de mensagens, há uma certa similaridade com as características dos sistemas robóticos móveis. Na troca de mensagens, os processos enviam e recebem mensagens da aplicação para que seja possível realizar uma computação, o mesmo ocorrendo em sistemas robóticos, que podem ser escritos como sistemas distribuídos utilizando trocas de mensagens. Pretendeu-se explorar essas semelhanças durante o trabalho relatado nesta tese, buscando alternativas para solucionar problemas em nível de teste de integração de sistemas robóticos móveis.

2.5 Considerações finais

Neste capítulo foi apresentada uma visão geral sobre a atividade de teste de *software*. Os fundamentos da atividade de teste foram apresentados no início do capítulo, com a elucidação das principais técnicas de teste (funcional, estrutural e baseada em defeitos). Também foi apresentado em mais detalhes o teste de integração, objetivo de estudo desta tese de doutorado. O teste de programas concorrentes também foi tratado, cujos conceitos foram utilizados durante o desenvolvimento desta tese de doutorado. O próximo capítulo apresenta conceitos de sistemas robóticos móveis.

SISTEMAS ROBÓTICOS MÓVEIS

3.1 Considerações iniciais

Neste capítulo são apresentados conceitos sobre sistemas robóticos móveis. É apresentado um paradigma muito utilizado para desenvolver esse tipo de sistema, o *publish/subscribe*, e o *middleware* Robotic Operational System (ROS), que vem sendo amplamente utilizado nesse domínio para o desenvolvimento de sistemas.

Este capítulo está organizado como se segue. Na Seção 3.2 são descritos conceitos sobre sistemas robóticos móveis. Na Seção 3.3 são detalhados aspectos do desenvolvimento de aplicações robóticas e é apresentado o *middleware* ROS com um exemplo de uso. Na Seção 3.4 estão as considerações finais deste capítulo.

3.2 Robôs móveis

Robótica é a ciência que percebe e atua no mundo físico por meio de dispositivos computacionais (THRUN *et al.*, 2006). Há diversos tipos de robôs, dentre eles os robôs móveis. Uma característica dos robôs móveis que os difere dos robôs industriais, por exemplo, é que eles possuem uma plataforma capaz de grande mobilidade no ambiente (ar, terra ou água), que serve como base para qualquer tarefa a ser executada. O princípio básico de um robô móvel é que ele seja capaz de navegar de uma posição conhecida para uma nova posição, evitando obstáculos. Aplicações robóticas móveis vêm sendo utilizadas em diferentes áreas, por exemplo, em aplicações domésticas (são exemplos aspiradores de pó e cortadores de grama), industriais (como exemplos, transporte automatizado e veículos de carga autônomos), urbanas (cadeiras de rodas robotizadas e veículos autônomos), militares (como exemplos, sistemas de monitoramento aéreo remoto - VANTs e sistemas táticos e de combate) de segurança e defesa civil e militar (exemplos são os robôs de patrulhamento de ambientes, resgate e exploração em ambientes

hostis). Essa diversidade de aplicações dos robôs móveis mostra o interesse econômico envolvido no desenvolvimento e aplicação desses sistemas (WOLF *et al.*, 2009).

Para interagir com o mundo físico os robôs móveis necessitam de dispositivos sensoriais, chamados sensores, capazes de “sentir” o mundo ao redor e dispositivos responsáveis pelos movimentos, que são os atuadores. Os sensores mais comuns são utilizados para detecção de luz, som, ondas eletromagnéticas e contato. Exemplos são sensores infra-vermelho (IR), sensores do tipo Laser (LIDAR - *Light Detection and Ranging*), sistemas de GPS, odômetros, entre outros. Os atuadores mais comuns são baseados no motor DC ou motor de passo (*Step-motor*). Exemplos são rodas, esteira, juntas e articulações, hélices ou turbinas, entre outros.

Para que um robô móvel seja também autônomo as seguintes características devem estar presentes: 1) mobilidade para se locomover no ambiente; 2) raciocínio para transformar suas percepções e conhecimentos em ações; 3) percepção para perceber o ambiente e suas características e; 4) comunicação com um operador humano ou outros dispositivos. A robótica móvel inclui três tipos de autonomia: a) os robôs tele-operados, que precisam de um operador humano; b) os semi-autônomos, que possuem capacidade para realizar pequenas tarefas de forma autônoma, mais ainda precisam de um operador humano e; c) os robôs autônomos que não necessitam de operador humano para execução das tarefas para as quais foi projetado (DUDEK; JENKIN, 2010). Os robôs autônomos vêm despertando grande interesse e demanda, pois são capazes de operar em ambientes internos e externos, interagindo e tomando decisões para realizar tarefas com êxito sem intervenção humana (CARVALHO; KOWALTOWSKI, 2009).

Em razão dessas particularidades, os sistemas robóticos móveis devem apresentar alta qualidade, funcionalidade, confiabilidade e eficiência, que são as principais características a serem preservadas (EBERT; SALECKER, 2009; MARWEDEL, 2010). O primeiro passo para conseguir isso é utilizar uma metodologia de desenvolvimento que atenda essas condições e um ambiente de desenvolvimento que facilite a implementação desses sistemas.

3.3 Desenvolvimento de sistemas robóticos móveis

Desenvolver *software* para robôs é uma tarefa que inclui desafios em razão do crescente aumento na complexidade desses sistemas. Os vários tipos de robôs são dotados de diferentes plataformas de *hardware* e isso influi em um reúso de código, que não é tão trivial, pois é necessário que uma gama de testes sejam realizados para conferir se o componente introduzido não reduziu a qualidade do sistema. Além disso, como esta é uma área multidisciplinar, as arquiteturas de *software* devem também apoiar integração em grande escala (QUIGLEY *et al.*, 2009). O projeto de um sistema robótico envolve a especificação e seleção de diferentes componentes, sensores e atuadores, considerando suas particularidades e especificidades, e posteriormente a combinação desses itens em um sistema autônomo. O sistema a ser desenvolvido deve possuir dispositivos capazes de obter dados do ambiente (sensores), para que o sistema

de controle robótico inteligente possa planejar e acionar seus dispositivos de modo que o robô interaja com o ambiente e execute a ação esperada (WOLF *et al.*, 2009).

Como exemplo, o projeto do sistema de controle de um robô similar ao ilustrado na Figura 1 deve considerar vários aspectos, tais como (WOLF *et al.*, 2009):

Figura 1 – Robô Pioneer 3DX com sensores de sonar e laser rangefinder

Fonte: Wolf *et al.* (2009).

- Tipo de tarefa a ser executada pelo robô: patrulhamento e vigilância de uma instalação (indoor);
- Tipo e precisão dos sensores embarcados: Laser Sick LMS200, com capacidade de detectar obstáculos em um range máximo de 25 a 50m de distância (fecho laser estreito e direcional) com precisão de até 10mm (erro estatístico de 5mm), ângulo de abertura de 100 a 180 graus e tempo de resposta entre 10 e 50ms; Sonar (*ultrasonic transducer - range-finding sonar*) composto por um arranjo de 8 sensores dispostos de modo a ter uma cobertura frontal de 180 graus, com alcance variando de 15cm até aproximadamente 7m (detecta a reflexão do sinal do sonar propagado em forma de cone); câmera de vídeo com resolução de 640 x 480 pixels e varredura de até 20fps; encoders com precisão de 500-ticks usado para medidas de deslocamento (odometria);
- Tipo e precisão dos atuadores: acionamento diferencial (2 motores independentes), robô de características holonômicas (sem restrições de graus de liberdade, podendo girar e se deslocar em qualquer direção) e capaz de atingir velocidade de até 1.6 m/seg;

Com esse esquema pronto, no projeto do sistema robótico devem estar especificados como ocorrerá a fusão de sensores, visto que um único sensor capta apenas uma pequena parte do ambiente, sendo necessário que os dados de cada sensor sejam arranjados de forma a fornecer

uma visão apurada do ambiente. Também deve estar previsto o tratamento de erros para os atuadores, pois, muitas vezes, os erros inerentes à execução do *hardware*, devido a imprecisões podem alterar o resultado pretendido de ação. Dessa forma, o projeto do sistema de controle de um robô, como o apresentado na Figura 1, deve ser capaz de fornecer em parte ou todas as tarefas (WOLF *et al.*, 2009):

- Fusão de sensores: obter os dados a partir dos sensores, tratando limitações de alcance e precisão, por meio da fusão dos dados;
- Desvio de obstáculos: detectar obstáculos e evitar colisões;
- Auto-localização: localizar o robô no ambiente (fornecendo posição e orientação);
- Mapeamento do ambiente: explorar e construir o mapa do ambiente, identificando obstáculos;
- Planejamento de trajetórias: planejar e definir trajetórias a serem executadas pelo robô, especificando as ações, de modo a se deslocar de uma posição origem até uma posição objetivo;
- Planejamento de ações: estabelecer um plano de ações, como exemplos explorar o ambiente, seguir uma parede ou corredor, se deslocar em uma direção objetivo desviando de obstáculos, recolher objetos, seguir um comboio, patrulhar uma área, etc;
- Navegação: executar as ações planejadas de modo robusto, como por exemplo, executar o deslocamento de uma posição origem até uma posição objetivo, realizando ajustes durante o deslocamento para evitar colisões;
- Interação e comunicação: interagir e se comunicar com outros agentes presentes no ambiente. Como exemplo, o futebol de robôs, no qual múltiplos robôs devem coordenar suas ações a fim de atingir objetivos comuns.

Com vistas à execução dessas tarefas, os sistemas robóticos são geralmente desenvolvidos em duas fases: 1) implementação de algoritmos robóticos, fase em que os algoritmos são gerados e testados e; 2) integração do sistema na plataforma de *hardware*, em que os *software* é embarcado na plataforma objetivo. Alguns *middlewares* para o desenvolvimento de aplicações robóticas e sistemas complexos possuem diferentes comportamentos e especificidades, bem como qualidades únicas que tornam cada um mais indicado para uma determinada tarefa (FERNANDES *et al.*, 2014). Uma grande variedade de *middlewares* têm sido propostos, tais como (JUNG; DEGUET; KAZANZIDES, 2010): Player (GERKEY *et al.*, 2001), Orocos (BRUYNINCKX, 2001), Orca (BROOKS *et al.*, 2007), ROS (QUIGLEY *et al.*, 2009), Rock (JOYEUX; ALBIEZ, 2011), dentre outros.

O ROS ¹ é um *middleware* constituído por um conjunto de bibliotecas *open-source* e ferramentas, que resultam de um projeto cujo código é fruto de esforços da comunidade internacional de desenvolvedores de aplicações robóticas. *Middleware* é um programa que fornece serviços não existentes nos sistemas operacionais convencionais para as aplicações, atuando como uma camada entre a aplicação e o Sistema Operacional (SO). Em sistemas distribuídos um *middleware* permite a comunicação entre máquinas e dispositivos heterogêneos, na medida em que possibilita a cada máquina a execução de funcionalidades comuns de SOs, usando protocolos, como se todas as máquinas fossem iguais. ROS gerencia a mediação entre o SO e demais aplicações usando o modelo de comunicação *publish/subscribe*, serviços e *actions* para interação entre os nós do sistema distribuído. ROS é integrado com ferramentas e bibliotecas, tais como: openCV para processamento de imagens, PCL para processamento e visualização 3D *Point Cloud*, Gazebo para simulação virtual realística 3D e Player, que provê uma interface de rede para uma variedade de *hardware* robótico e sensores. Ele ainda fornece abstração de *hardware*, controle de dispositivos de baixo-nível, passagem de mensagens e gerenciamento de pacotes.

Um sistema construído no ROS consiste em um conjunto de nós que pode residir localmente ou estar distribuído entre diferentes máquinas, conectado em uma topologia de rede *peer-to-peer*. Um nó ou processo é um arquivo executável que usa o ROS para comunicar-se com outros processos. Nesta tese um nó de um programa distribuído robótico será chamado processo. Mensagens são tipos de dados em assinaturas e publicações em um tópico. Tópicos são meios para publicação e assinatura de mensagens. Um processo Mestre (*Master*) funciona como um serviço de nomes e gerencia os demais processos durante a execução. No modelo de comunicação *publish/subscribe*, processos podem publicar (escrever) em um tópico e/ou assinar (ler) informações de um tópico. Múltiplos *publishers* e *subscribers* concorrentes podem existir para um mesmo tópico e um processo pode publicar e/ou subscrever de múltiplos tópicos. Além do modelo de comunicação *publish/subscribe* ROS também inclui a comunicação usando serviços (*services*), protocolo síncrono definido como uma chamada de procedimento remoto, a qual espera por uma mensagem de resposta, sendo possível existir vários clientes simultâneos para um mesmo serviço. O ROS também disponibiliza *actions* que são um tipo específico de serviço, no qual o cliente obtém *feedbacks* do estado atual de um serviço solicitado. Para apoiar o desenvolvimento multilinguagem, ROS usa uma Interface Description Language (IDL) para descrever as mensagens enviadas entre os módulos.

Outro *framework* proposto para desenvolver aplicações robóticas é o Rock² (JOYEUX; ALBIEZ, 2011). Rock é um *framework* similar ao ROS e baseado no Orocos RTT (*Real Time Toolkit*). Rock foi inicialmente desenvolvido pelo DFKI Robotics Innovation Center³ e vem sendo utilizado para o desenvolvimento de aplicações robóticas reais. Rock usa scripts Ruby para

¹ <http://www.ros.org/>

² <http://rock-robotics.org/stable/>

³ <https://www.dfki.de/web>

inicializar os componentes de uma aplicação e monitorá-los durante a execução. Comparando o Rock e o ROS podem ser citados alguns pontos:

- ROS usa o modelo de comunicação baseado em tópicos enquanto Rock é baseado em conexão. O ROS ganha em *overhead*, porém, perde em flexibilidade com relação à dificuldade em controlar o fluxo de informações no sistema;
- ROS não utiliza um modelo formal, ao passo que os módulos do Rock são definidos usando uma descrição de modelo abstrata. Isso favorece a gestão de estados e conexões em tempo de execução;
- Como o Rock é baseado no Orocos RTT aplicações em tempo real podem facilmente ser implementadas diferentemente do ROS que requer interface especial para isso;
- Rock segue a tendência das bibliotecas maiores do ROS, como PCL e openCV, em separar bibliotecas e módulos, de forma que as camadas de comunicação e funcionalidade são separadas;
- Rock pode usar classes comuns C++ (com restrições) como tipos interface, enquanto que ROS requer um compilador especial para os tipos interface, favorecendo a comunicação entre módulos;
- Rock possui suporte nativo para a implementação de módulos que se dividem em processos menores para facilitar a comunicação. Em ROS isso também é possível com os *Nodelets*, mas não é tão transparente;
- ROS é mais utilizado do que Rock, possui uma biblioteca maior de módulos e ferramentas mais maduras.

O desenvolvimento de sistemas robóticos móveis deve imperativamente envolver um cuidadoso projeto tanto de *hardware* como de *software*. Um dos desafios da robótica é como integrar as informações vindas de todos os sensores, de modo a gerar comandos e controlar os diferentes dispositivos de atuação do robô, garantindo que a tarefa seja executada de modo esperado, sem colocar em risco tanto o robô quanto aqueles que o cercam, dado que um robô móvel possui um sistema embarcado crítico e assim, executa missões críticas (WOLF *et al.*, 2009).

3.3.1 Exemplo de um modelo gerado com o ROS

Para utilizar o ROS este deve estar instalado e com as variáveis de ambiente devidamente configuradas. Inicialmente, como estudo de caso para a abordagem proposta foi construída uma aplicação utilizando o ROS com o objetivo de familiarização com o *framework* e simulação da troca de mensagens de comunicação entre processos do sistema. A aplicação não teve

como objetivo ser funcional, apenas simular a troca de dados utilizando tópicos como canal de comunicação. Foram utilizados apenas tipos de dados primitivos e um grafo foi gerado para representação da aplicação, que pode ser observado na Figura 2.

Figura 2 – Grafo de comunicação para um carro robótico

Fonte: Elaborada pelo autor.

No grafo é possível visualizar os principais processos da aplicação se comunicando por meio dos tópicos *scan*, *vehicle_state*, *obstacles*, *velocity_commands*, *steering_commands*, *throttle_commands* e *brake_commands*. A comunicação por tópicos acontece com as trocas de mensagens entre os processos. Para os produtores e assinantes se comunicarem eles devem enviar e receber o mesmo tipo de mensagem. A comunicação é possível quando um processo publica e um ou mais processos assinam e recebem mensagens. Nessa aplicação, a todo momento mensagens estão sendo publicadas e recebidas. Um exemplo de comunicação ocorre, por exemplo, entre os processos *teleop_by_joy* e *cruise_control* quando o processo *teleop_by_joy* publica uma mensagem no tópico *velocity_commands*. Nesse caso, o processo *cruise_control*, que faz o controle do carro, acessa essa mensagem do tópico *velocity_commands*. Outras comunicações ocorrem entre outros processos da aplicação de forma paralela.

3.4 Considerações finais

Neste capítulo foram apresentados conceitos relacionados aos sistemas robóticos móveis. Foi apresentada também uma visão geral sobre o desenvolvimento de sistemas robóticos, destacando o *middleware* ROS com um exemplo de uso. No próximo capítulo serão apresentados estudos da literatura sobre teste em sistemas robóticos.

ESTADO DA ARTE SOBRE TESTE DE SOFTWARE PARA SISTEMAS ROBÓTICOS MÓVEIS

4.1 Considerações iniciais

Neste capítulo é apresentado o estado da arte sobre teste de *software* para sistemas embarcados e robóticos. Foram conduzidos dois estudos de mapeamento sistemático ao longo do doutorado. Um mapeamento sistemático é um tipo de revisão de literatura definido para construir um esquema de classificação no campo de interesse, com base nos estudos obtidos. Após a análise dos resultados é possível obter frequências de publicações por categorias, de forma a obter um mapa de publicações capaz de representar a área de pesquisa. Esse mapa é composto por evidências descritas em um alto nível de granularidade ([KITCHENHAM; CHARTERS, 2007](#)).

Dois estudos de mapeamento foram conduzidos, um em 2013, no início do doutorado, e outro em 2016. No primeiro mapeamento, a principal questão de pesquisa foi: quais abordagens de teste de integração para sistemas embarcados e robóticos têm sido desenvolvidas? No segundo estudo a questão de pesquisa foi alterada com o intuito de obter estudos mais específicos sobre sistemas robóticos, tornando-se: quais abordagens têm sido definidas para teste de *software* de sistemas robóticos? Após as seleções, foram obtidos nos dois mapeamentos 57 estudos.

Este capítulo está organizado da maneira a seguir. Na Seção [4.2](#) é descrita de forma resumida a metodologia utilizada para condução dos estudos de mapeamento. Na Seção [4.3](#) é apresentado um resumo dos estudos selecionados nos dois mapeamentos e que contribuem para o estado da arte neste domínio. Por fim, na Seção [4.4](#) são apresentadas as considerações finais deste capítulo.

4.2 Mapeamento Sistemático sobre teste de integração para sistemas robóticos

Com o objetivo de identificar estudos sobre teste de *software* para sistemas robóticos foram realizados dois estudos de mapeamento sistemático. O primeiro deles foi conduzido no início do doutorado (2013) e o objetivo era identificar estudos com abordagens de teste de integração aplicadas a sistemas críticos, incluindo sistemas embarcados, veículos autônomos e sistemas robóticos. Este estudo incluiu uma questão de pesquisa (*QP*) e três subquestões (*SQP_n*), assim elaboradas:

- *QP*: Quais abordagens de teste de integração para sistemas embarcados e robóticos têm sido desenvolvidas?
- *SQP₁*: Quais critérios de teste têm sido definidos para o teste de integração em sistemas embarcados e robóticos?
- *SQP₂*: Quais ferramentas de teste têm sido desenvolvidas para o teste de integração em sistemas embarcados e robóticos?
- *SQP₃*: Quais formas de validação vêm sendo conduzidas para abordagens de teste de integração em sistemas embarcados e robóticos?

O segundo estudo, foi conduzido no final do doutorado, em 2016, e a questão de pesquisa (*QP*) do primeiro mapeamento foi alterada para satisfazer aos objetivos do projeto de doutorado, com a necessidade de obter estudos sobre teste de software aplicado ao domínio de sistemas robóticos, sendo:

1. *QP₁*: Quais abordagens têm sido definidas para teste de *software* de sistemas robóticos?
2. *QP₂*: Como essas abordagens podem ser classificadas?

Para ambos os estudos foram selecionadas as bases de dados que indexam os principais congressos e periódicos da área de Engenharia de Software, sendo elas apresentadas na Tabela 1. Para construção das *strings* de busca foram utilizados os principais termos das questões de pesquisa e seus sinônimos, obtendo as *strings*:

((*integration test OR integration testing OR test level integration OR interface test OR interface testing OR component test OR components test OR component testing OR components testing*) AND (*embedded OR unmanned vehicles OR unmanned autonomous vehicles OR unmanned ground vehicles OR robot OR robotic OR uav*))

((“integration test” OR “integration testing” OR “software test” OR “software testing”) AND robot*)

Tabela 1 – Fontes eletrônicas utilizadas nas buscas.

Fonte	URL
Compendex	(www.engineeringvillage2.org)
IEEE Explorer	(ieeexplore.ieee.org)
Portal ACM	(portal.acm.org)
Science Direct	(www.sciencedirect.com)
Scopus	(www.scopus.com)
Web of Knowledge	(www.isiknowledge.com)

Fonte: Elaborada pelo autor.

Considerou-se a busca pelos termos da *string* constantes no título, resumo e palavras-chave dos estudos indexados pelas bases de dados. A seleção dos estudos foi realizada em duas fases: seleção preliminar e seleção final. Os critérios de inclusão (CI_n) e de exclusão (CE_n) foram:

- CI_1 : o estudo apresenta uma abordagem de teste aplicada a sistemas embarcados ou robóticos;
- CI_2 : o estudo está escrito em português ou inglês;
- CE_1 : o estudo descreve uma abordagem de teste de integração em outros contextos, como exemplo, circuitos integrados de *hardware*;
- CE_2 : o estudo trata da integração de sistemas e não do teste de *software* de integração;
- CE_3 : o estudo trata do teste de sistemas embarcados e robóticos usando somente simulação sem nenhuma técnica de teste associada;
- CE_4 : o estudo apresenta experimentos sobre a aplicação de uma abordagem de teste de integração existente.

Após a elaboração do planejamento teve início a fase de condução com a execução das buscas nas bases de dados. Após a seleção final foram obtidos para os dois estudos de mapeamento 57 estudos sobre abordagens de teste de *software* para sistemas embarcados e robóticos.

4.3 Análise dos resultados

Os estudos selecionados nos mapeamentos estão ilustrados na Tabela 2.

Tabela 2 – Estudos selecionados nos dois mapeamentos.

Mapeamento 2013	Mapeamento 2016
(HONKA; KATILAKOSKI, 1991), (LIMA; CAVALLI, 1997), (FIN; FUMMI; SIGNORETTO, 2001), (LEE; KIM; CHA, 2002), (PESONEN, 2006), (BUI; AKTOUF, 2007), (JUN; MOORE, 2007), (SEO <i>et al.</i> , 2007), (SUNG; CHOI; SHIN, 2007), (BELLI; HOLLMANN; PADBERG, 2009), (BRITO <i>et al.</i> , 2009), (FAN <i>et al.</i> , 2009), (PÉREZ; KAISER, 2009), (PÉREZ; KAISER, 2010), (PAKALA; KHAN; RAJU, 2010), (SAGLIETTI; PINTE, 2010), (SCHATZ; PFALLER, 2010), (DAMM <i>et al.</i> , 2011), (HURA; DIMMICH, 2011), (KANG; PARK, 2011), (LU <i>et al.</i> , 2011), (PAKALA; RAJU; KHAN, 2011), (SUNG <i>et al.</i> , 2011), (MATTIELLO-FRANCISCO <i>et al.</i> , 2012), (HOSSAIN; LEE, 2013)	(DOLIN; BENNETT; RIEFFEL, 2001), (SHARYGINA; PELED, 2001), (SCHERER; LERDA; CLARKE, 2005), (CHUNG; HWANG, 2007), (PLAKU; KAVRAKI; VARDI, 2007), (PETTERS <i>et al.</i> , 2008), (BIGGS, 2010), (GUNDERSON; GUNDERSON, 2010), (KANG <i>et al.</i> , 2010), (KIM; SON; KIM, 2011), (SZATMÁRI; OLÁH; MAJZIK, 2011), (KANG; PARK, 2012), (LUCAS; ELBAUM; ROSENBLUM, 2012), (POWELL <i>et al.</i> , 2012), (TSAI; HSIEH, 2012), (ARNOLD; ALEXANDER, 2013), (SATO, 2013), (YOON; KANG; PARK, 2013), (BIHLMAIER; WÖRN, 2014), (COOMBES <i>et al.</i> , 2014), (JOUNG <i>et al.</i> , 2014), (KOSTAL, 2014), (LEE; KANG; PARK, 2014), (LIER; LÜTKEBOHLE; WACHSMUTH, 2014), (LILL; SAGLIETTI, 2014), (WEISSHARDT <i>et al.</i> , 2014), (ANDREWS; ABDELGAWAD; GARIO, 2015), (ARAIZA-ILLAN <i>et al.</i> , 2015), (BRITO <i>et al.</i> , 2015), (ERNITS <i>et al.</i> , 2015), (JANSCHKE; MOROZOV, 2015), (MUNOZ <i>et al.</i> , 2015)

Fonte: Elaborada pelo autor.

Outras análises são apresentadas a seguir considerando os estudos provenientes dos dois mapeamentos. Com o intuito de fornecer uma visão da área de pesquisa ao longo do tempo, a Figura 3 apresenta os estudos por ano de publicação. Observa-se com o gráfico que essa é uma área de pesquisa relativamente nova, cujos estudos começaram em 1991. O crescimento maior acontece a partir de 2006 com uma queda em 2008 mas a partir de 2009 cresce novamente.

Buscando identificar as procedências dos artigos, a Figura 4 mostra os estudos organizados por fonte de publicação. Em sua maioria, eles são provenientes de conferências de áreas relacionadas, como sistemas embarcados, automação de sistemas, robótica, engenharia de *software* e teste. Os *journals* seguem a mesma linha, como eletrônica e microprocessamento, computação e sistemas, robótica, mecânica, engenharia de *software* e arquitetura naval.

Figura 3 – Número de estudos publicados por ano

Fonte: Elaborada pelo autor.

Figura 4 – Estudos por fonte de publicação

Fonte: Elaborada pelo autor.

Foi realizada uma análise visando identificar as colaborações entre os grupos de pesquisa nessa área. Na Figura 5 estão representadas as intersecções entre os grupos de pesquisa identificadas. Considerou-se como colaborações interações entre pesquisadores de diferentes departamentos da mesma universidade e de diferentes universidades ou instituições. Dos 57 estudos incluídos no mapeamento foram identificados 20 com algum tipo de colaboração entre os autores.

Respondendo às questões de pesquisa propostas para os mapeamentos, um gráfico de bolhas com os resultados obtidos é apresentado na Figura 6. Na parte central do gráfico, os estudos estão classificados pela técnica de teste utilizada. No lado direito, os estudos estão classificados em três categorias: proposição de critério de teste, método de geração de dados e abordagem de teste (quando não apresenta definição de critério ou método de geração de dados). Alguns estudos ocupam mais de uma dessas categorias, como [Belli, Hollmann e Padberg \(2009\)](#), [Mattiello-Francisco *et al.* \(2012\)](#), [Saglietti e Pinte \(2010\)](#), que estão em critério de teste e método de geração de dados. No lado esquerdo do gráfico, os estudos foram classificados segundo

Figura 5 – Colaborações entre os grupos de pesquisa

Fonte: Elaborada pelo autor.

o tipo de teste aplicado. Como ocorreu no lado direito, alguns estudos ocupam mais de uma categoria. Sharygina e Peled (2001), Araiza-Illan *et al.* (2015) estão em teste estrutural, baseado em modelos e teste de conformidade. Ernits *et al.* (2015) está em teste estrutural, baseado em modelos e cenários. Belli, Hollmann e Padberg (2009) está em teste estrutural, baseado em defeitos e teste de componentes.

Considerando o lado direito do gráfico, a maior parte dos estudos está na categoria abordagem de teste. Como essa área de pesquisa é multidisciplinar, as ênfases das abordagens de teste variam muito em seus objetivos, sendo esta categoria bastante heterogênea. Em sua maioria, os estudos propõem abordagens para testar o sistema embarcado ou robótico, sem necessariamente sistematizar a geração de dados; a preocupação é atender os requisitos, para o qual o sistema foi projetado. Para isso, as propostas incluem modelagens usando métodos

Figura 6 – Gráfico de bolhas para os estudos selecionados no mapeamento

Fonte: Elaborada pelo autor.

formais, *model checking*, teste funcional e estrutural. A maioria das abordagens são de teste funcional, fato compreensível visto que o foco maior é observar as entradas e saídas do sistema.

Das abordagens de geração de dados, teste baseado em modelos é muito utilizado. Acredita-se que essa seja uma preocupação cada vez maior na área de teste de *software* por permitir automatizar uma ação que antes era conduzida pelo testador de forma manual, custosa e propensa a erros. Em quantidades menores estão os estudos que apresentam critérios de teste. Dentre os estudos dessa categoria, um deles é um artigo com a proposta inicial dos critérios definidos durante este projeto de doutorado.

A Tabela 3 mostra quais estudos estão em cada uma das categorias do lado direito do gráfico de bolhas (Figura 6). Destacando que os estudos [Belli, Hollmann e Padberg \(2009\)](#), [Mattiello-Francisco et al. \(2012\)](#), [Saglietti e Pinte \(2010\)](#), fazem parte das categorias critério de teste e geração de dados e, assim, aparecem nos dois grupos.

Uma questão importante sobre os estudos está relacionada com a automatização das abordagens. A Figura 7 apresenta os estudos por forma de automatização. Em sua maioria, os estudos incluem definições de ferramentas ou *frameworks* de teste que permitem automatizar a abordagem proposta. Quando isso não ocorre, os autores automatizam uma parte da abordagem, utilizando algoritmos, componentes como *stubs* de teste ou *scripts*. Em número menor estão os estudos que utilizam ferramentas existentes para automatizar a abordagem, e, por fim, estão os estudos que não mencionam nenhuma forma de automatização.

A Tabela 4 complementa as informações apresentadas no gráfico de bolhas (Figura 6) mostrando os estudos por técnicas de teste, apresentados na parte central do gráfico.

Figura 7 – Automatização das abordagens propostas nos estudos

Fonte: Elaborada pelo autor.

Tabela 3 – Estudos selecionados.

Objetivo	Referência
Critério de teste	(BELLI; HOLLMANN; PADBERG, 2009), (SAGLIETTI; PINTE, 2010), (SUNG <i>et al.</i> , 2011), (MATTIELLO-FRANCISCO <i>et al.</i> , 2012), (BRITO <i>et al.</i> , 2015)
Geração de dados	(LIMA; CAVALLI, 1997), (DOLIN; BENNETT; RIEFFEL, 2001), (BELLI; HOLLMANN; PADBERG, 2009), (BRITO <i>et al.</i> , 2009), (KANG <i>et al.</i> , 2010), (PAKALA; KHAN; RAJU, 2010), (SAGLIETTI; PINTE, 2010), (SCHATZ; PFALLER, 2010), (KIM; SON; KIM, 2011), (LU <i>et al.</i> , 2011), (SZATMÁRI; OLÁH; MAJZIK, 2011), (KANG; PARK, 2012), (LUCAS; ELBAUM; ROSENBLUM, 2012), (MATTIELLO-FRANCISCO <i>et al.</i> , 2012), (POWELL <i>et al.</i> , 2012), (ARNOLD; ALEXANDER, 2013), (HOSSAIN; LEE, 2013), (YOON; KANG; PARK, 2013), (KOSTAL, 2014), (LILL; SAGLIETTI, 2014), (ANDREWS; ABDELGAWAD; GARIO, 2015), (ARAIZA-ILLAN <i>et al.</i> , 2015), (JANSCHKE; MOROZOV, 2015)
Abordagem de teste	(HONKA; KATTILAKOSKI, 1991), (FIN; FUMMI; SIGNORETTO, 2001), (SHARYGINA; PELED, 2001), (LEE; KIM; CHA, 2002), (SCHERER; LERDA; CLARKE, 2005), (PESONEN, 2006), (BUI; AKTOUF, 2007), (JUN; MOORE, 2007), (PLAKU; KAVRAKI; VARDI, 2007), (SEO <i>et al.</i> , 2007), (SUNG; CHOI; SHIN, 2007), (PETTERS <i>et al.</i> , 2008), (FAN <i>et al.</i> , 2009), (PéREZ; KAISER, 2009), (BIGGS, 2010), (GUNDERSON; GUNDERSON, 2010), (PéREZ; KAISER, 2010), (DAMM <i>et al.</i> , 2011), (HURA; DIMMICH, 2011), (KANG; PARK, 2011), (PAKALA; RAJU; KHAN, 2011), (TSAI; HSIEH, 2012), (SATO, 2013), (BIHLMAIER; WÖRN, 2014), (COOMBES <i>et al.</i> , 2014), (JOUNG <i>et al.</i> , 2014), (LEE; KANG; PARK, 2014), (LIER; LÜTKEBOHLE; WACHSMUTH, 2014), (WEISSHARDT <i>et al.</i> , 2014), (ERNITS <i>et al.</i> , 2015), (MUNOZ <i>et al.</i> , 2015)

Fonte: Elaborada pelo autor.

A seguir é apresentado um resumo de cada estudo primário segundo o tipo de teste mais relacionado aplicado, obedecendo a classificação do lado esquerdo do gráfico de bolhas, apresentado na Figura 6.

4.3.1 Teste de alcançabilidade

É uma abordagem de teste que combina teste determinístico e não determinístico (HWANG; TAI; HUANG, 1995). É baseado em uma técnica chamada teste baseado em prefixo, no qual o programa executa deterministicamente até um determinado ponto e depois deste ponto segue de forma não determinística, permitindo a execução de diferentes caminhos (LEI; CARVER, 2006).

Scherer, Lerda e Clarke (2005) propõem um método para testar *software* de controle baseado no Java PathFinder, um *model checker* desenvolvido na NASA Ames Research Center. A extensão apoia a modelagem de um escalonador de tempo-real e um sistema definido com

Tabela 4 – Estudos por técnica de teste.

Técnica	Referência
Teste estrutural	(DOLIN; BENNETT; RIEFFEL, 2001), (SHARYGINA; PELED, 2001), (PESONEN, 2006), (SEO <i>et al.</i> , 2007), (BELLI; HOLLMANN; PADBERG, 2009), (BIGGS, 2010), (LUCAS; ELBAUM; ROSENBLUM, 2012), (HOSSAIN; LEE, 2013), (KOSTAL, 2014), (LEE; KANG; PARK, 2014), (ARAIZA-ILLAN <i>et al.</i> , 2015), (BRITO <i>et al.</i> , 2015), (ERNITS <i>et al.</i> , 2015)
Teste baseado em defeitos	(BELLI; HOLLMANN; PADBERG, 2009), (SUNG <i>et al.</i> , 2011)
Teste funcional	(HONKA; KATTILAKOSKI, 1991), (FIN; FUMMI; SIGNORETTO, 2001), (CHUNG; HWANG, 2007), (SUNG; CHOI; SHIN, 2007), (PETTERS <i>et al.</i> , 2008), (LIM <i>et al.</i> , 2009), (PÉREZ; KAISER, 2009), (GUNDERSON; GUNDERSON, 2010), (KANG <i>et al.</i> , 2010), (PÉREZ; KAISER, 2010), (PAKALA; KHAN; RAJU, 2010), (KANG; PARK, 2011), (KANG; PARK, 2012), (POWELL <i>et al.</i> , 2012), (TSAI; HSIEH, 2012), (ARNOLD; ALEXANDER, 2013), (SATO, 2013), (YOON; KANG; PARK, 2013), (BIHLMAIER; WÖRN, 2014), (COOMBES <i>et al.</i> , 2014), (JOUNG <i>et al.</i> , 2014), (WEISSHARDT <i>et al.</i> , 2014), (MUNOZ <i>et al.</i> , 2015)
Teste baseado em modelos	(LIMA; CAVALLI, 1997), (SHARYGINA; PELED, 2001), (LEE; KIM; CHA, 2002), (SCHERER; LERDA; CLARKE, 2005), (BUI; AKTOUF, 2007), (JUN; MOORE, 2007), (PLAKU; KAVRAKI; VARDI, 2007), (BRITO <i>et al.</i> , 2009), (SAGLIETTI; PINTE, 2010), (SCHATZ; PFALLER, 2010), (DAMM <i>et al.</i> , 2011), (HURA; DIMMICH, 2011), (KIM; SON; KIM, 2011), (LU <i>et al.</i> , 2011), (PAKALA; RAJU; KHAN, 2011), (SZATMÁRI; OLÁH; MAJZIK, 2011), (MATTIELLO-FRANCISCO <i>et al.</i> , 2012), (LIER; LÜTKEBOHLE; WACHSMUTH, 2014), (LILL; SAGLIETTI, 2014), (ANDREWS; ABDELGAWAD; GARIO, 2015), (ARAIZA-ILLAN <i>et al.</i> , 2015), (JANSCHKE; MOROZOV, 2015), (ERNITS <i>et al.</i> , 2015)

Fonte: Elaborada pelo autor.

equações diferenciais. Para avaliação da abordagem foi utilizado o código-fonte de um robô que segue linhas no chão.

Plaku, Kavradi e Vardi (2007) apresentam um *framework* multicamadas para o teste de sistemas híbridos. Sistemas híbridos são modelos formais, que combinam dinâmica discreta e contínua com os modos de operação. Por exemplo, um sistema híbrido pode modelar um veículo em que a dinâmica de movimento varia dependendo das condições do terreno. Os sistemas são modelados usando autômatos que contêm estados seguros e não seguros, que são utilizados para compor um grafo de busca. Para avaliação da abordagem é utilizado um sistema robótico autônomo que demonstra que a abordagem é promissora.

Andrews, Abdelgawad e Gario (2015) apresentam uma abordagem para aplicar teste baseado em modelo para gerar dados de teste a partir do modelo de mundo que representa os atores de um sistema autônomo. O processo de geração inclui três fases: 1) modelar o mundo ativo construindo modelos de comportamento. Nessa fase são utilizados diagramas de classe UML e as interações são modeladas usando CEFSM (*Communicating Extended Finite State Machine*); 2) selecionar critérios de cobertura baseados em grafo para a geração de caminhos de teste e critérios de cobertura particionamento do espaço de entradas para dados de teste; 3) gerar AWTCs (*Abstract World Behavioral Test Cases*), que são caminhos de teste concorrentes, e então, gerar dados de teste para transformar esses caminhos de teste concorrentes em casos de teste executáveis. Foram definidas as técnicas de serialização de caminhos APSESCC (*All Possible Serialized Execution Sequences Coverage Criterion*) e RCC (*Rendezvous Coverage Criterion*) e um sistema autônomo foi modelado como exemplo de uso da abordagem proposta. Os resultados mostraram que RCC é alcançável e que o número de casos de teste executáveis depende do tamanho dos dados gerados e do número de atores envolvidos na execução.

4.3.2 Teste de conformidade

É uma abordagem de teste em que o objetivo é verificar se o comportamento do sistema em teste está em conformidade com a especificação.

Lima e Cavalli (1997) apresentam uma abordagem para gerar sequências de teste para componentes de sistemas embarcados. Testa-se o comportamento do sistema conforme a especificação. A abordagem proposta é baseada na: 1) definição de um procedimento de composição que permite a abstração dos sinais trocados entre os processos que compõem o sistema, enquanto preserva a interação entre o sistema e o ambiente e; 2) teste orientado a objetivo. As transições que refletem a especificação do comportamento do componente podem ser usadas para construir objetivos de teste que somente testam a implementação do componente. Esta abordagem apresenta as seguintes vantagens: não é necessário testar o sistema completo, é possível testar o comportamento do componente no contexto e detectar se a implementação do componente está em conformidade com a especificação. É também possível detectar se a implementação inclui um componente embutido, que é equivalente em contexto à especificação do mesmo.

Sharygina e Peled (2001) apresentam uma abordagem combinando teste de conformidade e teste baseado em modelos de verificação com o objetivo de usar uma ferramenta de teste para examinar os caminhos de execução do modelo, enquanto compara-os com o comportamento do código. Inicialmente é aplicado teste estrutural ao modelo para identificar defeitos na modelagem. *Model checking* é usado, além de outras ferramentas para métodos formais.

Damm *et al.* (2011) apresentam uma abordagem para teste de integração de componentes de um sistema embarcado usando o formalismo baseado em contratos. No artigo é utilizada a Linguagem de Especificação de Requisitos (RSL) para teste em nível de especificação. Um exemplo usando um sistema de freio é utilizado para ilustrar a abordagem.

Pakala, Raju e Khan (2011) exploram teste de integração entre componentes de processamento embarcados (EPCs), que são conectados por meio de portas de entrada e saída. Os EPCs são modelados usando MEFs determinísticas comunicantes, com o objetivo de identificar defeitos nas interfaces entre o *hardware* e o *software*. Um exemplo mostra que o teste de integração de dois subsistemas é uma abordagem promissora.

Araiza-Illan *et al.* (2015) propõem automatizar os testes baseados em simulação utilizando verificação dirigida a cobertura (CDV) para código robótico no contexto de interação robô-humano (HRI). Diferentes estratégias para geração de dados são consideradas, como pseudorandômica, pseudorandômica *constrained* e baseada em modelos para complementar umas às outras e explorar comportamentos não esperados do sistema. Foram implementados dois tipos de cobertura: código (declarações) e requisitos funcionais (asserções). Os autores concluem que a abordagem é escalável para sistemas mais complexos e pode ser utilizada para sistemas autônomos em geral.

4.3.3 Teste de componentes

Um componente pode ser definido como uma unidade com interfaces especificadas por meio de contratos e com dependências de contexto explícitas (SZYBERSKI, 2002). Um componente pode ser desenvolvido por terceiros e reusado em outras aplicações independentemente da linguagem de programação ou ambiente de desenvolvimento, reduzindo o tempo e custo de desenvolvimento. O teste baseado em componentes pode ser analisado segundo duas perspectivas (DELAMARO; MALDONADO; JINO, 2007): 1) do cliente, que são os desenvolvedores que utilizam os componentes em suas aplicações; 2) do desenvolvedor, que é quem implementa e testa o componente sem conhecimento de todos os tipos de aplicações que poderão usá-lo. Com base nisso, uma estratégia de teste de componentes deve compreender tantos testes unitários, realizados pelo desenvolvedor, quanto testes de integração, realizados pelo cliente. No caso do teste de integração o foco principal são as interfaces, pois estas conectam dois ou mais componentes quando a aplicação está em execução. Interfaces são pontos de interação dos componentes, por meio das quais um componente cliente usa os serviços declarados nas interfaces providas.

Honka e Kattilakoski (1991) apresentam uma abordagem de teste de sistemas embarcados de tempo real. Os autores apresentam um ambiente de teste capaz de simular interrupções, operações de entrada e saída e características do mundo real, necessárias aos testes de programas de tempo-real. O ambiente é composto por componentes, como janelas de simulação, que fazem a comunicação entre o ambiente e o sistema em teste. Os componentes do ambiente de teste executam como processos distintos, possibilitando conduzir teste de integração e teste de sistema. Os autores acreditam ser uma abordagem promissora.

Pesonen (2006) apresenta uma abordagem de teste para uma família de produtos usando aspectos. A abordagem de teste insere código de teste (usando aspectos) no sistema em teste por meio de *stubs*. A família de produtos é baseada no SO Symbian para dispositivos móveis. AspectC++ é usado para adicionar aspectos de acordo com a especificação funcional e comportamentos durante o teste. Aspectos *stubs* são usados junto com o módulo normal a ser testado para testes de integração. Um estudo de caso mostrou que a abordagem é promissora.

Bui e Aktouf (2007) propõem uma abordagem para diagnosticar defeitos na comunicação entre componentes. Nessa abordagem o System Under Test (SUT) é descrito como grupos de componentes em que é testada a comunicação entre eles. Modelos de estados são utilizados para modelar as entradas e saídas dos componentes. O conjunto de testes é inserido dentro dos componentes. Dois estudos de caso foram apresentados usando uma aplicação termostato, com dois algoritmos extraídos da literatura. Apesar de inicial os resultados são promissores.

Chung e Hwang (2007) aplicam teste de componentes de *software* Ubiquitous Robot Companion (URC). A abordagem proposta sugere utilizar cenários para o projeto de casos de teste e apresenta conceitos da norma International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC) 9126 para o teste de componentes de *software* URC.

Jun e Moore (2007) apresentam um *framework* de teste para apoiar teste unitário de componentes em sistemas distribuídos, construído utilizando Common Object Request Broker Architecture (CORBA), COM/.NET, J2EE/RMI. O *framework* coSpy instrumenta as aplicações usando um compilador IDL para inserir código de monitoramento nos *stubs* e *skeletons* gerados. Dados do monitoramento são utilizados para gerar um grafo de chamadas dinâmico que revela a comunicação entre os componentes. No grafo estão informações sobre o comportamento do sistema em execução, como parâmetros de entrada e saída, exceções ou alocação de recursos online, como latência, CPU, uso de memória virtual, número de threads, etc. Componentes *drivers* são conectados ao compilador IDL para execução dos dados de teste.

Seo *et al.* (2007) propõem uma abordagem de teste de interfaces. Nessa abordagem, são modeladas as interfaces de *hardware* e *software* do Sistema Embarcado (SE), considerando as diferentes camadas. É construído um modelo, parecido com o diagrama de classes UML, que mostra as interfaces do SE. Com base nesse modelo são gerados os dados de teste. Assim, são propostos também critérios de cobertura das interfaces. Para a execução dos testes são utilizados *scripts* e *breakpoints* nos locais do código que possuem declarações das interfaces sendo testadas.

Nesse artigo é proposta uma ferramenta de teste em C/C++, a Justitia.

Petters *et al.* (2008) apresentam estratégias de teste para *software* de controle para times de robôs autônomos. Os autores sugerem utilizar dois *frameworks*, RoboFrame (PETTERS; THOMAS; STRYK, 2007) e MuRoSimF (FRIEDMANN; PETERSEN; STRYK, 2008), para condução de testes de componentes, testes *online* e testes *offline*.

Belli, Hollmann e Padberg (2009) apresentam uma abordagem de teste de integração utilizando o Grafo de Sequência de Comunicação (GSC), expressando fluxo de controle e fluxo de dados. Com base no grafo são definidos critérios de cobertura e adicionadas expressões booleanas para representar estruturas de dados complexas e comunicação entre os componentes. Operadores de mutação foram definidos para introduzir defeitos relacionados à inserção ou remoção de nós e arestas no GSC. Os grafos mutantes auxiliaram na avaliação das sequências de teste geradas com base no grafo. Um estudo de caso utilizando um sistema de controle de robô ilustra a abordagem, que precisa ser avaliada para sistemas de médio e de grande porte.

Pérez e Kaiser (2009) propõem uma abordagem para reúso de casos de teste em nível de integração, tentando resolver o problema de atrasos introduzidos quando são reusados casos de teste de componentes para teste de integração. O atraso dos balanceadores leva ao atraso em seqüências de sincronização. Um compensador suprime o atraso global antes da avaliação do teste. Um exemplo usando um sistema de controle automático de luz é usado para validar a abordagem, que mostra-se como uma alternativa à parametrização considerando atrasos nos componentes.

Biggs (2010) apresenta um método para teste de regressão de *software*, que faz interface com o *hardware* robótico. O método usa *logs* da biblioteca de comunicação Flexiport, que é uma biblioteca de comunicação orientada a *stream* escrita em C++. Ele recupera uma seção de comunicação entre o *hardware* e o *software driver*, que depois pode ser reproduzida, emulando o *hardware* e permitindo o *software* ser testado de forma independente do *hardware*.

Kang *et al.* (2010) propõem uma ferramenta de automação de teste de interfaces de componentes de sistemas robóticos. Ela gera e executa os dados de teste em ambiente de simulação. A ferramenta utiliza uma representação das interfaces, por exemplo, descritas usando uma IDL ou eXtensible Markup Language (XML). Após isso, um *parser* extrai dessa representação informações sobre parâmetros de entrada e saída do componente objetivo. Quando os valores pertencem a um intervalo, o gerador pode utilizar os critérios funcionais particionamento em classes de equivalência ou análise do valor limite para gerar os dados, a saída esperada é inserida pelo usuário. A ferramenta gera o código de teste que é executado.

Pérez e Kaiser (2010) tratam do reúso de casos de teste em vários níveis usando critérios de teste funcional. É apresentado um exemplo de um controle de luz para ilustrar a abordagem e resultados de um experimento usando quatro projetos de diferentes tamanhos demonstrando a escalabilidade do reúso *bottom-up* e casos de teste multicamadas.

Pakala, Khan e Raju (2010) propõem uma abordagem de teste de integração de sensores e/ou atuadores usando MEFs comunicantes e teste funcional. Modelos de sensores baseados em MEFs determinísticas são propostos e defeitos são inseridos por meio do canal de comunicação. A importância de testes funcionais e reuso no teste de integração é discutida. Um exemplo ilustra a aplicação de teste funcional para integração de sensores com EPC.

Hura e Dimmich (2011) propõem uma abordagem de teste usando diagramas de sequências para geração de dados de teste parcialmente automatizada. Máquinas de estados são construídas para verificar as sequências geradas com os diagramas de sequências (*logs*). O código-fonte do programa é instrumentado, compilado e embutido em um microcontrolador e com a execução do SUT as sequências são avaliadas pelo analisador de sequências (MEFs) e são gerados os relatórios do teste.

Kang e Park (2011) propõem uma abordagem de teste baseada em especificação de interfaces de componentes de *software* robóticos (RPIST). O método propõe um algoritmo para gerar dados de teste de forma manual, usando teste funcional e uma arquitetura para execução dos testes, usando *stubs*. Para a geração dos dados, por meio de um modelo de interfaces, são usados os critérios funcionais particionamento de equivalência e análise do valor limite. Um estudo de caso, usando um componente de *software* robótico, mostrou que a abordagem é promissora.

Lu *et al.* (2011) propõem uma abordagem em que os dados de teste são gerados com base em componentes AADL. Architecture Analysis and Design Language (AADL) é um tipo de linguagem de projeto arquitetural baseada em Arquitetura Dirigida a Modelos (MDA). Na abordagem proposta, um modelo de teste de componentes (CTM), que é uma máquina de estados finita, é construído a partir do modelo de componentes do sistema AADL. Para revelar defeitos relacionados às transições, uma árvore de teste é construída com base em CTM. Os nós e arestas da árvore representam estados e transições no CTM. Um estudo de caso utilizando um sistema de controle de voo é utilizado na validação da proposta. Atualmente uma ferramenta de teste está sendo desenvolvida para apoiar a abordagem.

Sung *et al.* (2011) propõem um conjunto de operadores de mutação para testar as interfaces de um sistema embarcado. São exploradas as interfaces entre o SO de tempo real embarcado e a aplicação e entre o SO e o *hardware*. Para inserir defeitos no código foi conduzido um estudo usando três premissas: i) quando? ou o momento em que o defeito pode ser inserido; ii) onde? ou o local do código em que o defeito pode ser inserido e; iii) como? ou de que forma introduzir uma alteração. Com base nessas premissas, são definidos três operadores de mutação: alteração do valor de variáveis globais, remoção de conexões de *hardware* e alteração na voltagem do dispositivo. Os operadores definidos devem ser aplicados para testar todo o sistema depois que o *software* é integrado com o *kernel* e dispositivos de *hardware*. Um estudo de caso é conduzido para validação da proposta.

Kang e Park (2012) propõem um *framework* de teste funcional automático para componentes de *software* robótico. O *framework* fornece serviços de teste para diferentes plataformas

robóticas, ambiente de teste distribuído e compartilha recursos de teste. A arquitetura do *framework* é composta por um servidor de teste (WebATES), um agente de compilação dos testes (TBA), um servidor que gerencia a configuração do *software* (SCMS) e um simulador. O gerador de dados de teste utiliza critérios de teste funcional para gerar os dados com base na especificação. Após a execução um relatório do teste é gerado.

Lucas, Elbaum e Rosenblum (2012) apresentam uma abordagem de teste de componentes que analisa sequências e frequências de mensagens dos componentes de um sistema distribuído robótico para identificar falhas. Um *driver* de teste é desenvolvido para definir as sequências de mensagens e execução simbólica dinâmica é utilizada para gerar caminhos e dados de teste. A abordagem é implementada no contexto do ROS.

Mattiello-Francisco *et al.* (2012) propõem uma abordagem denominada InRob, para a construção de modelos baseados em serviços, com o objetivo de gerar dados de teste de interoperabilidade e robustez. Nessa abordagem, o SUT é modelado usando autômatos de entrada e saída temporais e modelos parciais para evitar o problema da explosão de estados. Com base nos autômatos e formalismos *Test Purpose* (TP) e *Intermediate Format* (IF) são gerados dados de teste. Um estudo de caso utilizando um sistema embarcado de um satélite é conduzido para a validação da abordagem.

Hossain e Lee (2013) propõem uma abordagem de geração de dados de teste para resolver problemas de dependência, considerando definição e uso de variáveis globais e gerar dados de teste, de acordo com sequências de teste possíveis. É utilizada execução simbólica para gerar os dados de teste de acordo com os caminhos de teste. Na abordagem apresentada o número de sequências de teste é reduzido comparado a outras abordagens como teste randômico ou teste randômico orientado a caminhos.

Lee, Kang e Park (2014) propõem uma ferramenta para teste de interfaces baseada em simulação. Essa ferramenta fornece funções para gerenciar a documentação do projeto e automatizar a execução do teste de componentes de *software* robótico. A ferramenta foi desenvolvida como *plugin* do Eclipse.

Lier, Lütkebohle e Wachsmuth (2014) apresentam uma abordagem de teste para robôs autônomos, consistindo em ambiente de simulação, um *framework* de teste baseado em máquinas de estado finito e um servidor de integração contínua (DUVALL; MATYAS; GLOVER, 2007) para centralizar a avaliação dos resultados. *Scripts* MATLAB, R ou Python podem ser utilizados na avaliação. Com os testes em simulação os *logs* são analisados e podem ser alterados para executar novos testes.

Weisshardt *et al.* (2014) apresentam uma plataforma de teste para componentes de *software* robótico. A plataforma é embutida no servidor de integração contínua Jenkins¹ e oferece diferentes tipos de teste, como: teste em simulação (*simulation-in-the-loop*), *hardware*

¹ <http://jenkins-ci.org>

(*hardware-in-the-loop*) e teste de cenários. Inicialmente, o código é carregado no Jenkins a partir dos repositórios, em seguida o código é compilado. Em ambiente de simulação são fornecidos dados de entrada ou dados de *log*. Podem ser utilizadas as seguintes métricas: tempo para que o robô navegue de uma posição ao outra, distância, rotações e número de colisões. Para os testes de *hardware* cada atuador é avaliado. Vários componentes podem ser avaliados usando cenários. Ao final a ferramenta fornece os relatórios do teste.

Apenas um dos estudos propõe critérios de cobertura para teste de integração (BRITO *et al.*, 2015), o qual é fruto deste trabalho de doutorado. O artigo contém um estudo inicial dos critérios de teste definidos nesta tese.

Janschek e Morozov (2015) reportam uma metodologia para automatizar o teste de sistemas de controle de robôs industriais complexos (CIRs). Utiliza um modelo matemático baseado em restrições para gerar sequências de teste e testar aspectos de tempo de sistemas de controle distribuídos para CIRs. O modelo é implementado em um ambiente de integração contínua. A avaliação empírica mostra que a abordagem é apropriada para gerar sequências de teste de forma automática para CIRs e pode ser implantada no contexto industrial.

4.3.4 Teste de cenários

Cenários especificam possíveis usos ou ações do sistema a partir de diferentes usuários. No teste de cenários são utilizados cenários para testar o sistema (KANER, 2003). O teste de cenários visa garantir então que as funcionalidades do sistema em teste estão funcionando como esperado.

Lee, Kim e Cha (2002) apresentam uma abordagem para gerar dados de teste a partir de diagramas de sequência de mensagens (MSCs) construindo máquinas de estado finito global (GFSM) equivalentes. Foi desenvolvida uma ferramenta para editar MSCs e gerar automaticamente dados de teste. Os usuários descrevem as funcionalidades requeridas em cenários e os dados de teste são automaticamente gerados a partir do GFSM, de acordo com critérios de cobertura de estados e transições. A abordagem foi aplicada a um *software* embarcado para TV digital (DTV) desenvolvido pela Samsung. 20% das funcionalidades foram testadas e o sistema em teste facilitou a redução dos estados do modelo por possuir uma entrada bem específica. O estudo de caso mostrou que a abordagem é escalável para outros sistemas industriais.

Brito *et al.* (2009) propõem uma abordagem de teste de integração para arquiteturas de *software* tolerantes a faltas usando como modelos formais *B-Method* e *Communicating Sequential Process* (CSP). Um cenário é uma sequência de eventos esperados durante a operação do sistema, isto é, os cenários estão relacionados ao comportamento do sistema para implementação das funcionalidades. Para gerar os dados de teste a abordagem usa a especificação formal dos elementos arquiteturais. As sequências de solicitações e respostas envolvendo as interfaces são representadas por grafos de interação construídos usando a notação de diagramas de atividade

UML e a partir destes são gerados os dados de teste.

Ernits *et al.* (2015) propõem uma abordagem de teste de integração de *software* de robôs móveis construídos em ROS. Inicialmente é aplicado teste baseado em modelos usando *Uppaal timed automata (TA)* (BENGTSSON; YI, 2004) e adaptadores das saídas do modelo para o robô. São gerados modelos a partir de mapas topológicos e cenários são definidos a partir de sequências de estados. Os resultados são avaliados usando cobertura de código.

4.3.5 Teste de desempenho

O objetivo é mostrar que um sistema atende aos requisitos com relação ao processamento das transações e tempo de resposta. Para conduzir estes testes, normalmente são utilizados *scripts* de teste e uma infraestrutura para automatização dos testes. Algumas métricas são utilizadas: 1) disponibilidade, que é o tempo que o sistema está disponível para os usuários; 2) tempo de resposta, que é o tempo necessário para que o sistema responda aos pedidos do usuário; 3) *throughput*, que é a taxa de ocorrência dos eventos do sistema; 4) utilização, que é a porcentagem da capacidade de um recurso que está sendo usado (MOLYNEAUX, 2009).

Gunderson e Gunderson (2010) apresentam uma metodologia de teste de desempenho para robôs de serviços. Os autores discutem o uso de métricas de desempenho para o teste de sistemas para robôs autônomos inteligentes. É utilizada a métrica tempo total como a medida de tempo para que o robô atinja um objetivo desde o ponto inicial. Dessa forma, conduzem teste funcional.

Joung *et al.* (2014) usam ferramentas *Computational Fluid Dynamics (CFD)* para conduzir testes de resistência para prever a força de arrasto em um robô de disco subaquático. Uma ferramenta de teste foi desenvolvida para apoiar a aplicação da abordagem.

Munoz *et al.* (2015) apresentam uma ferramenta para teste de desempenho de *software* de controle de robôs autônomos. São utilizadas as seguintes métricas: tempo operacional (tempo que o controlador gasta para atualizar seu estado interno e escalonar objetivos), tempo de processamento do objetivo, tempo de processamento de estados e tempo de deliberação. O *framework* OGATE permite criar diferentes cenários e configurações, mas para isso o usuário deve implementar as interfaces requeridas para controlar e inspecionar partes relevantes do controlador autônomo e fornecer um arquivo de configuração XML, incluindo dados do controlador, descrição de cenários e métricas para os testes. OGATE é implementada em Java e comunica-se com o controlador por Transmission Control Protocol (TCP)/Internet Protocol (IP).

4.3.6 Teste de rede

O objetivo deste tipo de teste é identificar problemas que podem ocorrer no canal de comunicação. As abordagens nesta categoria não necessariamente conduzem teste de rede, mas utilizam a infraestrutura da rede para o teste.

Fin, Fummi e Signoretto (2001) apresentam uma abordagem de teste de integração para *IP-cores* (componentes). Nessa abordagem são usados *stubs* para interfacear com os *sockets*, pois a abordagem simula uma comunicação cliente servidor. Considerou-se três abordagens de implementação dos componentes: SystemC, VHDL e mista. São utilizadas MEFs para a geração de dados e para detecção de defeitos são realizadas três formas de simulação: local, local com *socket*, remota na intranet e remota na internet. Um exemplo é apresentado para validação da abordagem.

Satoh (2013) apresenta um *framework* para testar a mobilidade de robôs, por exemplo robôs de transporte industriais. O *framework* permite testar aspectos de *software* em rede como controle, descoberta de serviços, comunicação e outros. Foram implementados emuladores e uma máquina virtual. Os emuladores funcionam como agentes móveis, podendo carregar e testar o *software* como se tivesse sido movido com o computador em que estava sendo executado, conectado a serviços em sua rede local. A máquina virtual executa em diferentes máquinas sem desconectar o cliente ou a aplicação. A experiência inicial com a implementação do protótipo sugere uma redução no tempo de desenvolvimento e teste para computadores industriais em rede.

Coombes *et al.* (2014) descrevem o desenvolvimento de um sistema para teste de veículos autônomos, utilizando infraestrutura de rede convencional para comunicação e controle. O sistema utiliza ArduPilot, que é um piloto comercial automático baseado em Arduíno, que permite o controle remoto do veículo via comandos do protocolo MAVLink por meio de comunicação serial. Modems seriais *wireless* ponto-a-ponto são utilizados para o envio de mensagens. Um exemplo de uso é apresentado utilizando um quadrimotor.

4.3.7 Teste de robustez

Robustez pode ser definida como o nível em que um sistema ou um componente pode funcionar corretamente na presença de entradas inválidas ou condições ambientais degradantes, como temperatura, pressão, radiação, ou outros. Assim, o teste de robustez pode ser considerado sob estes dois pontos de vista: entradas inválidas e condições degradantes do ambiente (POWELL *et al.*, 2012).

Powell *et al.* (2012) propõem um método e uma plataforma para teste de robustez de mecanismos relacionados a propriedades de segurança da camada funcional de um sistema autônomo estruturado hierarquicamente. É utilizado teste funcional, de forma que os dados de teste são gerados por mutação aleatória de sequências de entradas válidas em *scripts* de teste. Um analisador de traces é utilizado para analisar a execução. A abordagem foi aplicada a camada funcional do Dala rover, um robô que é atualmente usado no LAAS (*Laboratory for analysis and architecture of systems* em Toulouse, França, para experimentos de navegação.

4.3.8 Search-based software testing

É a aplicação de técnicas de busca metaheurísticas para resolver problemas em teste de *software*, verificação e validação. O critério de adequação de teste é transformado em uma função *fitness* e um conjunto de soluções no espaço de busca é avaliado com relação a função *fitness* usando técnicas de busca metaheurísticas (AFZAL; TORKAR; FELDT, 2009). Um problema pode ser resolvido usando uma estratégia como busca aleatória, busca local (exemplo, *hill climbing*, *simulated annealing* e busca tabu), algoritmos evolucionários (exemplo, algoritmos genéticos, estratégias de evolução e programação genética), otimização de colônias e *particle swarm* (ROSHAN; PORWAL; SHARMA, 2012).

Dolin, Bennett e Rieffel (2001) exploram a seleção de dados de teste para *software* de controle distribuído para robôs modulares utilizando *search-based*. Cada caso de teste é uma configuração do mundo como um movimento que o robô deve fazer para atingir um objetivo. O espaço de casos de teste é extremamente grande e uma função de treinamento é desenvolvida para obter uma amostra efetiva desse conjunto. Os autores comparam diversas abordagens como coevolucionária, amostragem randômica e amostragem fixa. Para o exemplo usado, a amostragem randômica superou a taxa de amostragem fixa e obteve resultados similares às técnicas coevolucionárias consideradas.

Saglietti e Pinte (2010) apresentam técnicas para teste de integração de sistemas baseados em componentes. Em nível de teste estrutural foi desenvolvida uma ferramenta que utiliza algoritmo genético para gerar dados. Em nível de teste *grey-box* são utilizadas máquinas de estados comunicantes para modelar a comunicação entre os componentes. Com base nesse modelo são definidos critérios de cobertura de interfaces baseados nos estados, transições, efeitos e *triggers* para o caso particular de comunicação de dois componentes. Uma ferramenta de teste foi desenvolvida para apoiar a aplicação desses critérios utilizando geração automática. Os resultados dos estudos de casos mostraram que a abordagem é promissora.

Szatmári, Oláh e Majzik (2011) apresentam uma abordagem para geração de dados de teste para agentes de *software* autônomos utilizando teste funcional. O método usa metamodelos do ambiente derivados a partir de uma ontologia baseada em contexto e utiliza especificações das entradas e saídas para definir os testes. A abordagem sugere o uso de técnicas de busca metaheurística para a geração de dados de teste ou *search-based*.

Lill e Saglietti (2014) apresentam uma abordagem para testar o comportamento cooperativo de agentes autônomos. A abordagem inclui três fases de teste e a utilização de critérios de teste baseados em Redes de Petri Coloridas e um método de geração de dados que utiliza algoritmos genéticos. A abordagem apresenta uma forma de sistematizar o teste de sistemas autônomos cooperativos.

4.3.9 Teste de sistema

No teste de sistema, o objetivo é verificar se as funcionalidades foram implementadas corretamente, se os requisitos não funcionais satisfazem a especificação e características como correção, coerência e completude também são avaliadas (DELAMARO; MALDONADO; JINO, 2007). A aplicação é testada com outros sistemas de *software* e elementos de *hardware* que possivelmente estarão utilizando-o após sua liberação.

Schatz e Pfaller (2010) apresentam uma abordagem de teste de integração de componentes. Para a geração de dados é usado *weak second order monadic structure with one successor function (WSIS)*. Esse formalismo é apoiado pelo *model checker* Mona. Usando WSIS o comportamento é especificado por predicados com base em traces, que são sequências de estados. Os testes gerados são a nível de sistema para testar componentes.

Kim, Son e Kim (2011) propõem uma abordagem de teste que gera dados com base em diagramas de estado e executa-os em ambiente de simulação. Para isso, foi desenvolvida uma ferramenta para apoiar o procedimento, que foi aplicada a um sistema robótico articulado. Os resultados mostraram a eficácia da abordagem em revelar defeitos na fase de projeto do sistema.

Arnold e Alexander (2013) apresentam uma abordagem de geração de dados de teste para algoritmos de controle de robôs autônomos. A abordagem utiliza *Procedural Content Generation (PCG)* para a geração de cenários de teste. Segundo ela, o primeiro passo é gerar mapas do ambiente, depois um gerador de missões atribui o número de robôs e rotas a esse ambiente. Um monitor acessa dados da execução durante a simulação e depois analisa o *log* pra produzir relatórios do teste.

Yoon, Kang e Park (2013) propõem uma estratégia de combinação de dados de teste para a geração de dados com foco em *software* robótico. O método é chamado *Advanced ParaOrder (APO)* e gera suítes de teste mínimas. Uma comparação com outras estratégias de combinação é apresentada para um *software* robótico.

Bihlmaier e Wörn (2014) apresentam uma metodologia baseada em tecnologias existentes para teste de *software* chamada *Robot Unit Testing (RUT)*. O objetivo é estender o teste automatizado de componentes de *software* para o robô completo. RUT testa todos os componentes de *software* incluindo as interfaces com o simulador.

4.3.10 Testbed

As abordagens de teste identificadas nessa categoria são diversificadas. *Testbed* consiste em utilizar uma plataforma de *hardware* para execução dos testes, ou seja, os testes consideram além do simulador o suporte de uma plataforma de *hardware* capaz de fornecer mais evidências sobre o comportamento do *software*.

Sung, Choi e Shin (2007) propõem uma abordagem para teste de interfaces entre o

hardware e o SO do sistema embarcado (SE). Os componentes do SE são classificados em categorias, como HDS (*software* dependente de *hardware*), ADS (*software* dependente da aplicação), etc. Itens de teste são características do SE a serem testadas e com base nelas são definidas 24 interfaces em nível de *hardware* e 108 em nível de *software*. É apresentado um estudo empírico em que foram utilizadas 8 aplicações submetidas a revisão de código e teste funcional. Os resultados indicam que o modelo pode detectar defeitos nas interfaces de *hardware* e SO, enquanto o SE é carregado e executado. Uma ferramenta de teste está sendo desenvolvida para apoiar a abordagem proposta.

Fan *et al.* (2009) propõem um modelo e um *framework* de teste automático para componentes de *software* do tipo Robot Technology Component (RTC) e módulos de *hardware*. Com base no *modelo V*, o modelo de teste consiste em teste de unidade, integração e sistema. Os três níveis de teste correspondem a teste de *hardware*, teste da Application Programming Interface (API) do *hardware* e teste de sistema. O teste da API do *hardware* testa a interoperabilidade do módulo de *hardware* com os componentes de *software*. Os dados de teste são gerados a partir de critérios de teste funcionais. Sinais *wireless* enviam informações do *software* embarcado para a plataforma de teste que controla o *hardware*. Experimentos são aplicados para avaliar o modelo de testes utilizando um robô com um sensor ultra-sônico. Resultados mostram que a abordagem é eficiente em termos de tempo e custo.

Tsai e Hsieh (2012) apresentam um sistema de teste automatizado para robôs humanóides, que são robôs com aparência humana, utilizam duas pernas para locomoção e fazem movimentos similares aos humanos. Os itens de teste são classificados em quatro grupos: estrutura, movimento, visão e acústica. Cada grupo possui uma estação de teste para os subitens, como cabeça e corpo para estrutura, ou caminhar e virar a direita para movimento, dentre outros. Diagramas de sequências UML são usados para descrever procedimentos de teste dos itens.

Kostal (2014) apresenta uma abordagem de teste para um robô industrial automobilístico. Na abordagem, ao código fonte de teste são inseridos comandos extras para testar movimentos de precisão do robô não previstos nos testes iniciais. O *framework* desenvolvido encontra no *log* de teste os novos movimentos e permite reprodução dos testes.

4.4 Considerações finais

Neste capítulo foram apresentados os resultados de dois estudos de mapeamento sistemático conduzidos ao longo do doutorado, com o objetivo de obter estudos primários sobre teste de sistemas embarcados e robóticos.

Alguns *gaps* foram identificados, como a definição de critérios de cobertura para sistemas dessa área de pesquisa, fato explorado durante este doutorado com o intuito de fornecer uma abordagem de teste de integração capaz de melhorar a sistematização da atividade de teste nesse contexto. Outra lacuna apontada nestes estudos de mapeamento e explorada nesta tese

de doutorado foi a definição de abordagens de teste para aplicações desenvolvidas utilizando o *framework* ROS. Dos estudos incluídos no mapeamento, seis apoiam a definição de abordagens de teste em programas implementados utilizando o ROS (LUCAS; ELBAUM; ROSENBLUM, 2012; WEISSHARDT *et al.*, 2014; BIHLMAIER; WöRN, 2014; LIER; LütKEBOHLE; WACHSMUTH, 2014; ARAIZA-ILLAN *et al.*, 2015; BRITO *et al.*, 2015). Todos estes estudos foram publicados a partir de 2012, ano que teve início este doutorado.

Outras contruibuições dos mapeamentos foram: 1) identificação de um programa exemplo no estudo de Lucas, Elbaum e Rosenblum (2012), o qual foi utilizado como exemplo de uso durante a definição da abordagem de teste proposta; 2) utilização de cenários, que foi revista nos estudos Weisshardt *et al.* (2014), Ernits *et al.* (2015), Chung e Hwang (2007), Brito *et al.* (2009), Lee, Kim e Cha (2002), fato que evidenciou os benefícios da utilização de cenários em uma abordagem de teste para este domínio; 3) definição de grafos gerados a partir do código-fonte foi revisitada em Jun e Moore (2007), Belli, Hollmann e Padberg (2009) para auxiliar na definição do PSCFG. Além dessas contribuições soma-se o conhecimento adquirido na área de pesquisa com a condução dos estudos de mapeamento.

Um terceiro estudo de mapeamento foi conduzido durante o doutorado com o objetivo de fornecer evidências sobre abordagens para geração de dados para teste de integração. O teste de integração tem particularidades diferentes do teste de unidade e abordagens específicas são necessárias. Este estudo de mapeamento sistemático foi publicado no ESELAW 2016, para maiores detalhes consultar Brito *et al.* (2016). No próximo capítulo será apresentada a abordagem de teste proposta nesta tese de doutorado.

TESTE DE INTEGRAÇÃO BASEADO NO MODELO DE COMUNICAÇÃO PUBLISH/SUBSCRIBE

5.1 Considerações iniciais

Nos capítulos anteriores foram apresentados conceitos relacionados ao teste de *software* e sistemas robóticos móveis. A atividade de teste aplicada a sistemas robóticos móveis possui desafios ainda não solucionados, pois, características como não determinismo das entradas e comunicação entre os componentes aumentam a complexidade do teste e devem ser consideradas. Apesar desses desafios, abordagens definidas para o teste de programas sequenciais e concorrentes podem ser adaptadas para o domínio de sistemas robóticos compreendendo a proposição de critérios de cobertura e ferramentas de teste. No contexto desta tese, considera-se o teste de sistemas robóticos em relação à comunicação entre os processos ou componentes do sistema.

Este capítulo apresenta um modelo de teste para o problema abordado nesta tese: aplicar teste de integração a sistemas robóticos móveis. A solução para o problema consiste em desenvolver um modelo de teste de integração capaz de revelar defeitos em sistemas robóticos. O modelo deve ser capaz de utilizar informações da especificação e do código-fonte do programa e, para tanto, são utilizadas as técnicas de teste funcional e estrutural. Neste trabalho é proposta uma extensão ao modelo definido em [Souza *et al.* \(2008\)](#), [Souza, Souza e Zaluska \(2014\)](#) para teste estrutural de programas concorrentes, adicionando detalhes do modelo de comunicação *publish/subscribe*, como fraco acoplamento entre os processos.

Assim, a Seção [5.2](#) descreve as similaridades entre os programas concorrentes e os sistemas robóticos móveis considerados nesta tese de doutorado. A Seção [5.3](#) sumariza os defeitos comuns para o modelo de comunicação *publish/subscribe* para sistemas robóticos. A Seção [5.4](#) apresenta conceitos do teste de integração com base no modelo de comunicação

publish/subscribe para sistemas robóticos. A Seção 5.5 detalha o modelo definido para representar o comportamento das aplicações, incluindo a geração de cenários de teste e os critérios de cobertura propostos neste trabalho com um exemplo de aplicação. Por fim, na Seção 5.6 estão as considerações finais deste capítulo.

5.2 Comparação entre programas concorrentes e o modelo de comunicação *Publish/Subscribe*

Ao iniciar as pesquisas deste doutoramento o primeiro passo foi identificar e compreender os pontos de concorrência nas aplicações robóticas alvo para este projeto, com o intuito de aproveitar os conhecimentos do teste de programas concorrentes e adaptá-los para aplicações robóticas, quando necessário. A comparação de forma qualitativa sobre as similaridades entre os dois contextos apresentada nesta seção baseia-se em aplicações robóticas escritas utilizando o modelo de comunicação *publish/subscribe* implementado no ROS e os programas concorrentes aqui referenciados são os que utilizam passagem de mensagem MPI.

Nos programas sequenciais as instruções e comandos são executados sequencialmente, de maneira que, enquanto uma instrução está sendo executada há subutilização de recursos, pois, outras instruções poderiam estar sendo executadas no mesmo instante de tempo em outras partes do *hardware*, aproveitando recursos ociosos do processador. A programação concorrente surgiu como alternativa a esse problema e tem como objetivo otimizar o desempenho das aplicações, permitindo uma melhor utilização dos recursos, quando um ou mais processos interagem entre si para resolver um problema. Dessa forma, um programa concorrente é formado por vários processos, tarefas ou *threads* que se comunicam durante a execução do programa. Um processo é um programa em execução, formado por um programa executável, seus dados, o contador de instruções, os registradores e todas as informações necessárias à sua execução (TANENBAUM, 2001).

As aplicações robóticas aqui estudadas podem ser representadas como um grafo de nós, os quais representam processos no *middleware* ROS. Os nós podem residir em uma mesma máquina ou em máquinas diferentes, dado que estes compõem um sistema distribuído. Aplicações robóticas são projetadas de modo a possuírem mecanismos capazes de fornecer os dados necessários para que o sistema de controle robótico inteligente possa planejar e realizar o acionamento dos seus dispositivos executando a ação esperada. Sistemas robóticos possuem muitos componentes que precisam operar concorrentemente como os sensores, atuadores, módulos de locomoção e manipuladores, que são combinados em um sistema autônomo. Essa característica é que aproxima as aplicações robóticas das aplicações concorrentes.

Ambos os tipos de aplicações realizam comunicação entre processos (programas concorrentes) ou nós (programas robóticos) por meio da passagem de mensagem e fazem uso da concorrência. Os pontos de sincronização em um programa concorrente permitem que os

processos se comuniquem recebendo ou enviando dados de uns para os outros. Uma operação bloqueante síncrona (envio utilizando *MPI_Ssend* no MPI, por exemplo) implica que o processo que transmite a mensagem seja bloqueado até que receba uma confirmação de recebimento da mensagem pelo processo receptor. Esse tipo de comunicação deve ocorrer quando o processo emissor necessita garantir a recepção antes de continuar a sua execução. Como desvantagem, está a possibilidade de atrasar a execução, pois, enquanto o processo receptor não recebe a mensagem, o processo emissor poderia estar executando outras atividades. Caso contrário, se tem uma operação não bloqueante assíncrona (envio utilizando *MPI_Bsend* no MPI, por exemplo), isto é, o processo transmissor envia a mensagem (que deve ser armazenada em um *buffer*) e continua sua execução. O processo emissor não fica dependente da sincronização com o processo receptor, e pode continuar sua execução. Essa forma de envio tem a vantagem de não requerer sincronização, mas é necessário definir explicitamente um *buffer* associado ao programa. Um erro pode ocorrer caso não exista espaço suficiente no *buffer*. Uma operação de envio com bloqueio em modo pronto (*MPI_Rsend* no MPI, por exemplo) é iniciada se uma operação de recebimento combinada tiver sido executada. No caso de operação com bloqueio, a única espera na tarefa emissora é que todos os dados tenham sido transferidos do *buffer* da mensagem. No envio com bloqueio no modo padrão (*MPI_Send* no MPI, por exemplo), a implementação do MPI decide se a mensagem ficará pendente no ambiente de execução durante algum tempo (colocadas em um *buffer*) ou não, e, nesse caso, ocorre sincronização. As quatro formas de comunicação são amplamente utilizadas em programas concorrentes que usam passagem de mensagem.

Nas aplicações robóticas, em um nível de abstração mais alto, esses pontos de sincronização ocorrem nas interfaces dos nós. Nesse caso há uma ligeira diferença, pois nas aplicações robóticas há basicamente três tipos de comunicação: baseada em tópicos, baseada em serviços e *actions*. Na comunicação utilizando tópicos, há um *buffer* entre o nó *publisher* que publica dados naquele tópico e o nó *subscriber* que subscreve dados daquele tópico. Os tópicos possibilitam uma comunicação não bloqueante assíncrona, que implementa um fluxo de dados e troca de mensagens, possibilitando que múltiplos *publishers/subscribers* concorram a um único tópico e a ordem das mensagens é irrelevante. Essa característica possibilita um fraco acoplamento entre os nós do sistema, pois torna o nó *publisher* desconectado do nó *subscriber*. Na comunicação utilizando serviços, a comunicação é bloqueante síncrona, definida como uma chamada de procedimento remoto, a qual exige uma mensagem de resposta, admitindo a existência de vários clientes ao mesmo tempo. Na comunicação utilizando *actions* a comunicação é não bloqueante e síncrona. Nesse caso um nó envia um pedido a outro nó, porém, o nó que solicitou a informação continua sua execução normalmente.

A concorrência existe a partir do momento em que dois ou mais processos iniciaram sua execução, mas não terminaram (ALMASI; GOTTLIEB, 1994). Nos programas concorrentes há a presença do não determinismo, o qual não permite saber antecipadamente quais sincronizações ocorrerão entre os processos. Na execução de um sistema robótico, a partir da inicialização a execução ocorre de forma concorrente e dois tipos de não determinismo podem ocorrer: não

determinismo nas entradas, as quais são obtidas por meio dos sensores (espera-se que uma faixa de valores seja obtida) e o não determinismo durante o processamento das chamadas *callback*, quando utilizada a comunicação usando tópicos, as quais são manipuladas por *threads* internas ao ROS. Além disso, a execução das aplicações robóticas se caracteriza por um alto nível de concorrência entre os módulos de comunicação (nós) que controlam e monitoram os sensores e atuadores dos robôs, e o sistema de controle ([ARAIZA-ILLAN et al., 2015](#)).

Um sistema computacional distribuído, conforme [Tanenbaum \(2001\)](#) e [Coulouris et al. \(2001\)](#), é uma coleção de computadores autônomos, interligados por uma rede de comunicação e equipados com um SO distribuído, que permite o compartilhamento de recursos existentes no sistema, isto é, para o usuário final não é visível a existência de múltiplos recursos. As aplicações escritas no contexto do ROS são por si só distribuídas, isto é, cada nó do sistema pode estar localizado em qualquer local da rede em que o sistema está conectado, com algumas exceções, por exemplo, um nó *driver* que se comunica com um *hardware* deve rodar na máquina em que o *hardware* está fisicamente conectado. Um sistema executando em ROS pode comportar dezenas ou centenas de nós rodando em várias máquinas distintas. A depender das configurações qualquer nó pode necessitar se comunicar com qualquer outro nó do sistema a qualquer momento.

Conhecendo-se as principais diferenças entre os dois tipos de programas, é possível traçar um paralelo sobre as diferenças no teste dos dois tipos de programas. Algumas abordagens de teste de programas concorrentes são provenientes do teste de programas sequenciais, em que alguns métodos foram adaptados para comportar características de programas concorrentes, como a concorrência e o não determinismo (Seção 2.4). Abordagens para o teste de aplicações robóticas também precisam de adaptações que comportem características específicas desse tipo de aplicações como o fraco acoplamento entre os nós, a concorrência e o não determinismo.

No contexto do *middleware* ROS, algumas abordagens de teste vêm sendo desenvolvidas. O teste de componentes visando identificar sequências de mensagens específicas e frequências em que uma falha pode ocorrer é apresentado por [Lucas, Elbaum e Rosenblum \(2012\)](#). Nessa abordagem é usada execução simbólica dinâmica para gerar testes capazes de exibir falhas nos componentes dos sistemas testados. Em [Weisshardt et al. \(2014\)](#) os autores propõem uma plataforma de teste para pacotes de *software* utilizando o servidor Jenkins ([KAWAGUCHI; CROY, 2017](#)). A abordagem consiste em teste de *hardware*, em que cada componente é testado, e em teste de simulação, utilizando métricas como duração do teste, distância percorrida, quantidade de rotações e número de colisões para avaliar um componente. Uma metodologia denominada teste de unidade robótico (RUT) é apresentada em [Bihlmaier e Wörn \(2014\)](#), que explora o fluxo de dados entre sensores e atuadores, inclui código para implementar os casos de teste, modelos de simulação, atuadores e sensores simulados com interfaces apropriadas.

[Lier, Lütkebohle e Wachsmuth \(2014\)](#) propõem uma cadeia de ferramentas que compreendem um ambiente de simulação, um *framework* para inicializar automaticamente um sistema, verificar a execução dos componentes a partir de experimentos e um servidor de integração

contínua para avaliação dos testes. O teste é conduzido por meio de *waypoints* enviados aos agentes em execução. É utilizado um framework de teste baseado em máquina de estado finito (FSMT), que é executado incluindo simulação no servidor de integração contínua. [Araiza-Illan et al. \(2015\)](#) apresentam uma abordagem de verificação composta por um *testbench* que possui gerador de teste, *driver*, *checker* e avaliação de cobertura. Diferentes tipos de modelos podem ser utilizados nesse teste, como cobertura de código (declarações), cobertura de requisitos (asserções) e outros.

Essas abordagens de teste exploram em sua maioria o comportamento de um componente e quando é avaliado mais de um componente é conduzida uma variante de teste funcional ou teste baseado em especificação. A próxima seção apresenta os defeitos mais comuns desse tipo de sistema.

5.3 Defeitos comuns em robôs móveis

Uma taxonomia de falhas para robôs móveis é apresentada em [Carlson, Murphy e Nelson \(2004\)](#). Nesse estudo, uma falha é definida como a incapacidade do robô ou equipamento usado com o robô funcionar normalmente. As falhas são categorizadas em dois grupos: físicas e humanas (Figura 8).

Figura 8 – Taxonomia de falhas para veículos terrestres não tripulados

Fonte: Adaptada de [Carlson, Murphy e Nelson \(2004\)](#).

As falhas físicas são subdivididas em tipos relacionados às plataformas robóticas, são eles: efetoras (*effector*), sensor, sistema de controle, energia e falhas de comunicação. Efetores são definidos como quaisquer componentes que executam atuação e/ou relacionamentos com esses componentes. Nessa categoria estão incluídos, por exemplo, motores, garras e rodas. Falhas do tipo sensor estão relacionadas a problemas com os dispositivos utilizados para perceber o ambiente (sensores). A categoria sistema de controle inclui o computador de bordo, *software* do fabricante e quaisquer unidades de controle de operação remota (do inglês *operator control units- OCU*). As falhas relacionadas à energia consideram problemas de bateria ou na fonte

de alimentação dos dispositivos. Falhas de comunicação estão relacionadas com a perda da capacidade do robô se comunicar com outros robôs, dispositivos ou humanos. As falhas humanas são subdivididas em falhas de projeto e de interação. As falhas de interação podem ser de dois tipos: erro e engano. Segundo essa taxonomia, erros são originados de ações conscientes do programador, como por exemplo, um mal-entendido sobre a especificação do sistema que o fez implementar uma função incorretamente. Engano tem causas inconscientes, nas quais o programador tentou fazer uma ação correta mas não obteve sucesso. Neste caso, essas definições para erro e engano diferem da apresentada por [Delamaro, Maldonado e Jino \(2007\)](#) para teste de *software*, na qual um erro caracteriza-se por um estado inconsistente durante a execução do programa e um engano é uma ação humana capaz de produzir um defeito.

Cada falha, independentemente de ter causa física ou humana, possui dois atributos, reparabilidade e impacto. A severidade da falha é avaliada com base no impacto sobre a tarefa atribuída ao robô ou missão. Uma falha de robô terminal finaliza a missão atual do robô e uma falha não terminal introduz alguma degradação na capacidade do robô realizar a missão. A reparabilidade da falha é descrita como reparável de campo ou não reparável de campo. Uma falha é considerada reparável de campo se ela pode ser reparada em condições de ambiente favoráveis, com o equipamento que normalmente acompanha o robô no campo. Por exemplo, se um pequeno robô, que é transportado em uma mochila falha, as ferramentas necessárias para o reparo teriam que estar na mochila junto ao robô para que a falha seja classificada como reparável de campo ([CARLSON; MURPHY; NELSON, 2004](#)). No mesmo trabalho é apresentada uma análise de defeitos utilizando 13 aplicações robóticas, em que foram encontradas como falhas mais comuns as de efeito, sistemas de controle instáveis, plataformas projetadas somente para uma pequena gama de condições de funcionamento, alcance de rede sem fio limitadas e insuficiente largura de banda para transações envolvendo vídeo.

Uma taxonomia de ameaças que podem afetar um sistema autônomo é apresentada em [Lussier et al. \(2004\)](#). As ameaças consistem em falhas, erros e defeitos. Falhas de sistema são eventos que prejudicam a entrega do serviço correto. Um erro é definido como um evento não esperado que pode causar uma falha. Defeitos físicos ocorrem devido a fenômenos adversos, como exemplo, um sensor que retorna um valor não válido. Defeitos de projeto são causados de forma não intencional, durante o desenvolvimento do sistema. Defeitos de interação são resultantes da interação com outros sistemas ou usuários. Nesse grupo, há os defeitos de interação acidentais e os maliciosos. Um erro cometido por um programador é um exemplo de um defeito acidental, já um ataque intencional é um exemplo de um defeito malicioso ([LUSSIER et al., 2004](#)).

[Borenstein e Feng \(1996\)](#) e [Ribeiro \(1999\)](#) descrevem erros relacionados à odometria em robôs móveis. Dados de odometria são obtidos a partir da contagem de pulsos pelos *encoders*, que são sensores instalados nas rodas do robô. Os erros são divididos em dois grupos: 1) erros sistemáticos, que são originados de imperfeições do próprio robô, como exemplo as diferenças

no diâmetro das rodas e; 2) erros não sistemáticos são os erros originados da interação do robô com o ambiente, como exemplo escorregamento das rodas do robô em solos escorregadios.

Com base nas taxonomias apresentadas para sistemas robóticos em geral, em um contexto mais específico, em que são consideradas implementações que utilizam passagem de mensagem e o modelo de comunicação *publish/subscribe*, erros podem ocorrer em decorrência de diversas situações, tais como: 1) fila (*buffer*) de mensagens cheia; 2) não determinismo na rede de comunicação, que pode provocar atrasos; 3) falhas nos nós, quando um nó deixa de receber ou enviar dados; 4) mensagens perdidas; e 5) mensagens duplicadas ou corrompidas (SASNAUSKAS *et al.*, 2010). Esses tipos de erros estão relacionados tanto a causas humanas como físicas, como relacionado nas taxonomias Carlson, Murphy e Nelson (2004) e Lussier *et al.* (2004).

Durante o estudo dos defeitos típicos no contexto de robôs móveis observou-se a adoção de uma taxonomia de conceitos sobre erro e defeito diferente daquela empregada na comunidade de teste de software. Desse modo, para o restante desta tese e para o modelo de teste proposto serão consideradas as definições apresentadas por Delamaro, Maldonado e Jino (2007) e descritas na Seção 2.2.

5.4 Fundamentos do teste baseado no modelo de comunicação *Publish/Subscribe*

Em programas orientados a objetos e interativos as ações do usuário sobre os objetos, como manipulação de um botão do mouse ou entrada de texto via teclado, são tratadas como eventos que alteram outros objetos que mantêm o estado do programa. Um sistema baseado em eventos possui os seguintes elementos: eventos e notificações para a comunicação, produtores e assinantes interagindo, inscrições manifestando interesse do assinante em determinadas notificações e o serviço de notificação de eventos, que é o responsável por entregar notificações entre produtores e assinantes. Sistemas baseados em eventos distribuídos estendem o modelo de eventos locais permitindo a objetos em diferentes máquinas serem notificados sobre eventos que ocorreram em um outro objeto (COULOURIS *et al.*, 2001).

Publish/subscribe é um modelo de comunicação baseado em eventos implementado no ROS. ROS é baseado em eventos, e dessa forma, os nós assinantes são os consumidores e os que publicam são os produtores de eventos. Nesse sistema, um ou mais clientes registram suas funções *Callback* para um evento específico e quando o evento é gerado as funções são chamadas. Neste trabalho funções e métodos são termos intercambiáveis, para uma padronização é utilizado apenas o termo método. Os métodos *Callback* são executados quando uma interrupção ocorre, nesse caso a interrupção provoca a chegada de uma mensagem a um nó. A partir deste ponto um nó executado no ROS será denominado processo.

A comunicação usando o modelo *publish/subscribe* ocorre da seguinte forma: O processo

produtor serializa a mensagem a ser enviada em um *buffer* (fila) e retorna a execução. As *threads* internas do ROS acessam essa fila e distribuem as mensagens entre as filas dos assinantes correspondentes. Dessa forma, a comunicação explorada neste trabalho é assíncrona, dado que o processo produtor não aguarda que o envio do dado seja concluído para retornar a execução. Entende-se como envio do dado concluído a ação que possibilita que o processo assinante acesse o dado recebido. Além disso, a comunicação seria bloqueante se ela garantisse que o *buffer* com a mensagem (no espaço do usuário) pudesse ser reutilizado após a liberação do processo produtor sem que a mensagem fosse perdida ou sobreposta. Isso garantiria que o conteúdo da mensagem não seria corrompido pelo próprio processo transmissor após a execução do método *publish*. Porém, no ROS, se uma dessas filas ultrapassa a capacidade máxima, as mensagens mais antigas são descartadas, caracterizando assim, uma comunicação não bloqueante.

O ROS em tempo de execução constitui-se como uma rede de processos *peer-to-peer* que são fracamente acoplados utilizando a infraestrutura de comunicação do ROS. O fraco acoplamento resulta do fato dos publicadores e assinantes não se conhecerem e, dessa forma, há um desacoplamento entre a produção de informação e o seu consumo. Esse comportamento resulta de três características principais: 1) o serviço de eventos funciona como intermediário na comunicação entre os processos, de forma que o processo assinante não tem conhecimento sobre quem publicou determinado evento; 2) um evento é propagado de forma assíncrona para os assinantes que registraram interesse naquele evento, de forma que o assinante não precisa estar ativo no momento que o evento foi publicado e; 3) produtores não são bloqueados enquanto produzem eventos e assinantes podem ser notificados de forma assíncrona da ocorrência de um evento, enquanto estão realizando alguma atividade concorrente (EUGSTER *et al.*, 2003).

Em resumo, sistemas robóticos são compostos por um grande número de processos que interagem e concorrem por recursos e uma preocupação durante a atividade de teste é avaliar o impacto dessa característica nos resultados (BRITO *et al.*, 2009; LEE; KIM; CHA, 2002; LIMA; CAVALLI, 1997; LU *et al.*, 2011; PAKALA; KHAN; RAJU, 2010; PAKALA; RAJU; KHAN, 2011; DAMM *et al.*, 2011). Um defeito pode estar relacionado com a comunicação entre os processos e garantir a cobertura das diversas situações de comunicação por tópicos aumenta a confiança de que as interações entre as unidades são executadas corretamente. Para a aplicação desse tipo de análise de cobertura é necessário um modelo que represente o fluxo de controle e de dados de cada processo do programa em cada ponto de comunicação. O modelo de teste deve permitir identificar os dados de teste a serem executados e exercitar as possibilidades de comunicação.

Assim, este trabalho propõe uma abordagem de teste que utiliza cenários e define um grafo, inspirado no grafo de fluxo de controle (ALLEN, 1970), para sistemas robóticos móveis baseado no modelo de comunicação *publish/subscribe*. Os cenários de teste têm como objetivo testar as funcionalidades do sistema e, para isso, utiliza o critério funcional particionamento em classes de equivalência para auxiliar na geração dos dados de teste. A representação de um

programa P como um grafo de fluxo de controle $G = (N, E, s)$, para programas sequenciais, representa os blocos de comandos e desvios que indicam os possíveis caminhos de execução de P . N representa o conjunto de nós, E o conjunto de arestas, e s o nó de entrada. Um grafo de fluxo de controle é um grafo orientado, que possui um único nó de entrada $s \in N$ e um nó de saída $o \in N$, no qual cada vértice representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para outro. Cada bloco possui as seguintes características: 1) uma vez que o primeiro comando do bloco é executado, todos os demais são executados sequencialmente e; 2) não existe desvio de execução para nenhum comando dentro do bloco.

O grafo definido no contexto desta tese de doutoramento é inspirado no grafo de fluxo de controle, na medida em que representa blocos de comandos e desvios no fluxo de execução do programa. Porém, o grafo aqui definido pode possuir mais que um nó de entrada $s_i \in N$ e mais que um nó de saída $o_i \in N$, com $i \in \mathbb{N}$ e $i = 1, 2, \dots, k - 1$. Além disso, nem todos os blocos de comandos são representados no grafo, pois o objetivo do grafo é representar os pontos de integração do programa, ou seja, a comunicação entre os processos do sistema distribuído.

O grafo de fluxo de dados, também chamado grafo Def-Use foi proposto por [Rapps e Weyuker \(1985\)](#) como uma extensão ao grafo de fluxo de controle e representa informações do fluxo de dados do programa. Nesse grafo são representados pontos do programa em que é atribuído um valor a uma variável v , chamado de definição da variável, e pontos em que o valor de v é utilizado, chamados de uso da variável.

No grafo aqui definido, inspirado no grafo de fluxo de dados, são representadas as definições e usos de variáveis de comunicação ao longo da execução do sistema. Uma variável de comunicação é definida em um processo produtor e usada em um método *publish* durante a comunicação com um processo assinante. Com base nesse grafo é definida uma família de critérios de integração de fluxo de controle e de dados, os quais são apresentados na próxima seção.

5.5 Abordagem de teste para comunicação *Publish/Subscribe*

A abordagem de teste proposta do âmbito desta tese de doutorado constitui em uma abordagem híbrida, que aplica teste funcional e teste estrutural. O teste funcional é conduzido por meio da especificação de cenários de teste ([KANER, 2003](#)), utilizados para gerar dados de teste capazes de exercitar aspectos funcionais do sistema que se pretende testar. Os sistemas robóticos são sistemas baseados em eventos, assim, a especificação de cenários com base no comportamento do sistema em ambiente de simulação permite gerar dados de teste que exercitam comportamentos do sistema ajudando a selecionar dados de teste para o teste de cobertura.

O teste de cenários é baseado na especificação do sistema. O testador inicialmente especifica cenários de teste com base nas funcionalidades do sistema e, na sequência, são gerados dados de teste para satisfazer os cenários. No modelo de teste aqui proposto, para cada cenário pode ser gerado mais de um caso de teste. Para a geração dos dados de teste é aplicado o critério funcional particionamento de equivalência com base na especificação do sistema.

Além do teste de cenários, que tem como objetivo a geração de um conjunto de teste com base no comportamento do sistema, a abordagem inclui a definição de critérios de teste de integração baseados no modelo comunicação *publish/subscribe*. Para isso, é proposto um modelo de teste, representando o fluxo de controle e de dados do sistema em teste. Esse modelo é baseado nos trabalhos de Souza *et al.* (2008), Souza, Souza e Zaluska (2014), que definiram critérios de teste estrutural para programas concorrentes. O modelo definido neste trabalho adiciona detalhes da comunicação *publish/subscribe*, analisando o fraco acoplamento entre os processos.

Como parte do modelo, um Grafo de Fluxo de Controle baseado na Comunicação *Publish/Subscribe* ou Publish/Subscribe-based Control Flow Graph (PSCFG) é definido para representar um programa em teste *Prog*. O PSCFG consiste em uma junção de dois grafos: 1) o grafo de comunicação gerado pelo ROS, representando o conjunto de processos (P) e o conjunto de tópicos (T) do programa segundo o modelo de comunicação *publish/subscribe* e; 2) Grafos de Fluxo de Controle (GFCs) representando os conjuntos de nós (N^p) e os conjuntos de arestas (E^p) de cada processo do grafo de comunicação *ROS*, onde p é o i -ésimo processo do grafo de comunicação. Os GFCs são definidos neste trabalho para coletar informações sobre a comunicação entre os processos e não representam todos os possíveis fluxos de execução do sistema, como é utilizado no teste estrutural (nível de unidade) de aplicações tradicionais. Dessa forma, os mesmos representam somente as partes do código que se relacionam com a comunicação.

Assim, p é um processo que realiza computação e comunica-se com outros processos usando tópicos (modelo de comunicação *publish/subscribe*). O modelo considera um número n_p fixo e conhecido de processos criados durante a inicialização do sistema. Cada processo p_i é composto por um número n_m fixo e conhecido de métodos, definidos por $p_i = m_0^i, m_1^i, \dots, m_j^i$. Cada método m_x^p possui seu $GFC^{p,m}$, representando o fluxo de controle e variáveis de comunicação. A notação $n_i^{p,m}$ é utilizada para representar cada nó do $GFC^{p,m}$, sendo que n_i é a identificação do nó, p indica o processo e m o método que contém o nó. $N^{p,m}$ representa o conjunto de nós do processo p do método m . N_{pub} é o conjunto de nós que publicam mensagens, N_{sub} é o conjunto de nós que assinam mensagens e N_{com} é o conjunto de nós de comunicação, ou seja, nós que publicam mensagens e nós que assinam mensagens de outros nós.

As arestas de um PSCFG são assim classificadas:

1. conjunto de arestas *intramétodo* ($E^{p,m}$): contém as arestas internas a um processo p de um método m ;

2. conjunto de arestas *intermétodo* (E_i^p): contém as arestas entre dois métodos de um processo p ;
3. conjunto de arestas *intraprocessos* (E^p): contém as arestas internas a um processo p , formadas por $E^p = E_i^p \cup_{m=0}^{nm^p-1} E^{p,m}$;
4. conjunto de arestas *interprocessos* (E_i): contém as arestas de comunicação entre os processos, ou seja, arestas que representam os tópicos;
5. conjunto de arestas E : contém todas as arestas do grafo PSCFG, ou seja, $E = (E^p) \cup (E_i)$.

Um caminho *intramétodo* $\pi^{p,m}$ no $GFC^{p,m}$ é aquele que contém somente arestas $\in E^{p,m}$. O caminho $\pi^{p,m}$ é representado por: $\pi^{p,m} = (n_1^{p,m}, n_2^{p,m}, \dots, n_k^{p,m})$, em que $(n_i^{p,m}, n_{i+1}^{p,m}) \in E^{p,m}$. Um caminho *intermétodo* π^p no $PSCFG$ contém pelo menos uma aresta *intermétodo* e é representado por $\pi^p = (\pi^{p,m_1}, \pi^{p,m_2}, \pi^{p,m_3}, \dots, \pi^{p,m_n})$, de forma que toda aresta $(n_x^{p,m_k}, n_y^{p,m_l})$ faz parte do conjunto E_i^p . Um caminho *intraprocesso* é formado por um caminho *intramétodo* e/ou um caminho *intermétodo*. Um caminho *interprocesso* π no $PSCFG$ contém pelo menos uma aresta *interprocesso* e é representado por $\pi = (\pi^1, \pi^2, \dots, \pi^i, T)$, em que T é o conjunto de pares de comunicação entre processos, ou tópicos, que foram executados no caminho π , tal que $T \in E_i$.

Considerando o fluxo de dados, uma definição de variável ocorre quando tem seu valor armazenado em uma posição de memória. Nesse caso, a variável pode estar do lado esquerdo de um comando de atribuição ou ela pode ser passada por referência em algum método, armazenando o valor de saída ou o valor de retorno. V^p é o conjunto de todas as variáveis de comunicação do processo p . Para o modelo definido, as informações do código são extraídas de forma estática e, dessa forma, o modelo não mapeia variáveis criadas dinamicamente.

Um uso de uma variável v ocorre em um comando no qual o valor da variável é acessado. Diferentemente dos programas tradicionais, em que há definição de *c-uso* e *p-uso* para usos de variáveis, neste modelo, o interesse é apenas nas definições e usos de variáveis de comunicação. Um uso comunicacional (*m-uso*), ocorre em comandos de comunicação ou troca de mensagens no modelo *publish/subscribe* e está relacionado a arestas *interprocessos* $(n^{pk,mi}, n^{pl,mj}) \in (E_i)$ e variáveis de V^p .

Com base nessas definições a associação requerida pelos critérios é a *m-uso*, assim definida:

- *associação m-uso*: dada pela tripla $\langle n_i^{pk,mt}, (n_i^{pk,mt}, n_j^{pl,mw}), v \rangle$, tal que $\langle n_i^{pk,mt}, v \rangle$ é uma definição e $(n_i^{pk,mt}, n_j^{pl,mw})$ possui um *m-uso* de v e existe um caminho livre de definição com relação a v de $n_i^{pk,mt}$ para $(n_i^{pk,mt}, n_j^{pl,mw})$.

A associação *m-uso* pressupõe a existência de um método m_2 para a troca de mensagem em um processo diferente, caracterizando a comunicação *interprocesso*. O Quadro 1 sumariza os conceitos apresentados.

Quadro 1 – Resumo dos conceitos

Sigla	Conceito
P	conjunto de processos do grafo de comunicação ROS
T	conjunto de pares de comunicação do PSCFG (tópicos do grafo de comunicação ROS)
N	conjunto de nós
N^p	conjunto de nós de um processo p
$N^{p,m}$	conjunto de nós de um processo p de um método m
N_{pub}	conjunto de nós que publicam mensagens
N_{sub}	conjunto de nós que assinam mensagens
N_{com}	conjunto de nós de comunicação (nós que publicam e subscrevem mensagens)
E	conjunto de arestas
E^p	conjunto de arestas de um processo p
$E^{p,m}$	conjunto de arestas de um processo p de um método m (intramétodo)
E_i^p	conjunto de arestas entre dois métodos m_1 e m_2 de um processo p (intermétodo)
$\pi^{p,m}$	caminho intramétodo
π^p	caminho intermétodo
V^p	conjunto das variáveis de comunicação do processo p

Fonte: Elaborada pelo autor.

Com base no modelo de teste um conjunto de critérios de teste estrutural de integração para aplicações *publish/subscribe* é proposto.

5.5.1 Teste funcional: cenários de teste

Cenários têm sido muito utilizados na fase de elicitação dos requisitos de sistemas de *software* e durante o ciclo de desenvolvimento com o objetivo de auxiliar no entendimento dos requisitos, gerar dados de teste, facilitar a interação com os usuários do sistema, dentre outros. O teste de cenário é o teste baseado em um cenário (KANER, 2003). O testador especifica os cenários de teste para o sistema a ser testado com base nos requisitos e a partir dos cenários são gerados os dados de teste para exercitar aspectos funcionais do sistema.

Durante o desenvolvimento de sistemas robóticos são utilizados simuladores, que possibilitam a criação de cenários compostos por robôs, obstáculos e outros objetos, capazes de simular o comportamento do robô em um ambiente real. O modelo de teste definido nesta tese de doutorado utiliza teste de cenários com o objetivo de auxiliar na geração dos casos de teste. Para gerar os cenários de teste o testador deve utilizar as seguintes diretrizes (KANER, 2003):

- Identificar todos os cenários a partir da especificação do sistema;
- Para cada cenário, identificar um ou mais casos de teste.

Com os cenários definidos, o próximo passo é a geração dos dados de teste, utilizando o critério funcional Particionamento de Equivalência (MYERS *et al.*, 2004). Esse critério baseia-se na especificação do *software* para derivar os requisitos de teste. Ele foi selecionado por admitir entradas com diferentes tipos de dados, como normalmente são os sistemas robóticos,

em que estes, normalmente são dados provenientes de sensores, como sensores de proximidade, lasers, câmeras e outros. Para o modelo de teste aqui proposto, considera-se que a especificação de requisitos que gerou os cenários já foi avaliada e contém uma descrição completa das funcionalidades do sistema.

Para cada cenário observa-se o domínio de entrada, que é dividido em classes de equivalência válidas e inválidas, considerando as possíveis condições de entrada. Em seguida, seleciona-se o menor número possível de casos de teste, baseando-se na hipótese de que um elemento de uma dada classe seria representativo da classe toda. Um cenário de teste pode conter mais de um caso de teste, a condição de parada é a satisfação do cenário. A utilização do critério particionamento de equivalência é uma forma de sistematizar a geração dos dados de teste e reduzir o conjunto de dados de teste, começando pela seleção de casos de teste eficazes em exercitar funcionalidades do sistema.

Dependendo da necessidade ou criticidade do sistema, outros critérios funcionais podem ser aplicados, como Análise do Valor Limite (MYERS *et al.*, 2004), Teste Sistemático (LINK-MAN; VINCENZI; MALDONADO, 2003) ou outros.

5.5.2 Teste estrutural: critérios de teste

Com o objetivo de aplicar o teste de integração, exercitando características específicas da comunicação entre os processos os seguintes critérios de teste são definidos para exercitar o fluxo de controle:

- Critério todos-nós-comunicação (*all-nodes-communication criterion*): requer que todos os nós $\in N_{com}$ do *PSCFG*, ou seja, nós $\in N_{pub}$ e nós $\in N_{sub}$ sejam exercitados pelo menos uma vez pelo conjunto de teste.
- Critério todos-nós-publishers (*all-nodes-publishers criterion*): requer que todos os nós $\in N_{pub}$ do *PSCFG*, ou seja, nós que possuem uma chamada ao método *publish*, sejam exercitados pelo menos uma vez pelo conjunto de teste.
- Critério todos-nós-subscribers (*all-nodes-subscribers criterion*): requer que todos os nós $\in N_{sub}$ do *PSCFG*, ou seja, nós que possuem usos das mensagens recebidas, sejam exercitados pelo menos uma vez pelo conjunto de teste.
- Critério todos-pares-publishers-subscribers (*all-pairs-publishers/subscribers criterion*): requer que todos pares de nós, compostos por $n_i \in N_{pub}$ e $n_j \in N_{sub}$ do *PSCFG*, isto é, arestas interprocessos $e \in (E_i)$, sejam exercitadas pelo menos uma vez pelo conjunto de teste.
- Critério todos-k-pares-publishers-subscribers (*all-k-pairs-publishers/subscribers criterion*): requer que todos os caminhos $\pi = (\pi^1, \pi^2, \dots, \pi^i)$ contendo k pares de arestas inter-

processos $\in E_i$ do *PSCFG* sejam exercitados ao menos uma vez pelo conjunto de dados de teste. Para $k = 2$ o caminho a ser coberto possui no máximo 2 arestas interprocessos.

- Critério todas- k -mensagens (*all- k -messages criterion*): requer que todos pares de nós, compostos por $n_i \in N_{pub}$ e $n_j \in N_{sub}$, ou seja, arestas interprocessos $\in E_i$ do *PSCFG*, sejam exercitadas por uma sequência de k mensagens $\in$ ao conjunto de dados de teste. Para $k = 10$ são observadas 10 mensagens enviadas e recebidas pelos dois processos.

5.5.2.1 Exemplo de aplicação

Um sistema para um aspirador de pó, que é uma versão simplificada do iRobot Roomba¹, foi utilizado para ilustrar a aplicação do modelo de teste apresentado nesta tese de doutorado. Este sistema foi desenvolvido com base no trabalho de Lucas, Elbaum e Rosenblum (2012). O aspirador de pó é um robô autônomo capaz de explorar e mapear um ambiente desconhecido para limpá-lo sem nenhum tipo de comando por parte do usuário. Ele é dotado de um sensor de proximidade, para evitar colisão com obstáculos no ambiente e atuadores (rodas e aspirador) para que ele se desloque realizando a limpeza. Odometria é utilizada para auxiliar a estimar a posição do robô.

Os processos do sistema comunicam-se por passagem de mensagens armazenadas em filas (*buffers*) do ROS. A Figura 9 ilustra um diagrama para o robô aspirador de pó.

Figura 9 – Diagrama para o robô aspirador de pó

Fonte: Lucas, Elbaum e Rosenblum (2012).

Para simplificar o sistema considera-se as filas de mensagens de tamanho 1, permitindo a um processo assinante manter a próxima mensagem enquanto a mensagem atual está sendo processada. O processo *Controller* envia 1 mensagem contendo a velocidade para o processo

¹ <http://www.irobot.com/Company>

Collision_Avoidance em intervalos de tempo regulares. O processo *Collision_Avoidance* instrui o robô para reduzir a velocidade, se este está a 2 passos de tempo de um obstáculo e parar se o robô está a 1 passo de tempo de uma colisão. A instrução para o robô reduzir a velocidade é importante porque ele não pode parar em 1 passo de tempo, devido à inércia. Quando parado, o controlador rotaciona o robô até que o processo *Collision_Avoidance* retorne que o sistema está seguro e distante de obstáculos, dada a velocidade desejada e continua o processamento.

Com base na especificação, foram gerados os cenários de teste para o sistema, apresentados no Quadro 2.

Quadro 2 – Cenários de teste para o sistema

Cenário	Descrição do cenário
I	Verificar se o robô navega no ambiente com velocidade constante distante de obstáculos
II	Verificar se o robô navega no ambiente após uma falha no sensor de proximidade que o impossibilita de enviar dados
III	Verificar se o robô navega no ambiente após uma falha no sensor de proximidade que o faz enviar dados fora da faixa de valores esperados
IV	Verificar se o robô reduz a velocidade quando se aproxima de obstáculos
V	Verificar se após reduzir a velocidade o robô para para evitar colisão
VI	Verificar se após reduzir a velocidade o robô ele toma outro sentido com velocidade constante
VII	Verificar se após navegar com velocidade constante o robô mantém velocidade reduzida
VIII	Verificar se ao navegar com velocidade reduzida o robô adquire velocidade constante e continua com velocidade reduzida
IX	Verificar se após reduzir a velocidade o robô adquire velocidade constante e para para evitar colisão
X	Verificar se quando o robô está se deslocando com velocidade constante ele consegue parar

Fonte: Elaborada pelo autor.

Para a geração dos casos de teste o primeiro passo é elencar as entradas e saídas do sistema para cada cenário de teste. Quando em execução, o sensor de proximidade envia continuamente dados para o sistema. Uma entrada para quaisquer dos cenários aqui definidos é um dado ou uma sequência de dados fornecida pelo sensor de proximidade (ação que mais se aproxima de uma execução real do sistema). O sistema possui duas limitações: entradas pertencentes ao conjunto dos números naturais e intervalo de dados de início de uma sequência de entrada deve estar entre 9 e 127. Considerando que o sistema precisa de um tempo para os processos estarem todos ativos, a primeira mensagem de proximidade é perdida. Assim sendo, uma sequência de entrada deve conter dois dados iniciais de proximidade maiores que 9 e menores que 127 e, após isso pode conter quaisquer números maiores que 0. Com base nas entradas possíveis foram geradas as classes de equivalência com base no critério funcional particionamento de equivalência (Tabela 5), onde P é um valor fornecido pelo sensor, CV_i indica uma classe válida e CI_i uma classe inválida.

A outra parte do caso de teste refere-se à saída esperada. As possíveis saídas do sistema são: 1) robô se desloca com velocidade constante; 2) robô se desloca com velocidade reduzida; 3) robô para; 4) robô exibe mensagem de erro e; 5) robô colide com obstáculo. Com base nas classes de equivalência e saídas foram gerados os casos de teste ilustrados na Tabela 6. Esses

casos de teste satisfazem os cenários de teste e são utilizados como ponto de partida para a cobertura dos elementos requeridos gerados pelos critérios de teste na próxima fase do teste.

O código-fonte ilustrado no Apêndice A descreve os processos *Collision_Avoidance*, retirado de Lucas, Elbaum e Rosenblum (2012), e *Controller*, adaptado de Brito *et al.* (2015). Ao lado dos números das linhas do código estão os números dos nós do PSCFG, gerado com base neste código e processos mais simples que não estão representados no código.

Antes de construir o PSCFG é gerado o grafo de comunicação utilizando o ROS (Figura 10), que serve como base para a geração do PSCFG. Neste grafo, o sistema é representado por processos produtores e assinantes de conteúdo (elipses), conectados por tópicos (retângulos).

Figura 10 – Grafo de comunicação para o sistema do robô aspirador

Fonte: Elaborada pelo autor.

Os cinco processos *Controller*, *Collision_Avoidance*, *Proximity_Sensor*, *Motor_Driver* e *Mapping* comunicam-se usando os seguintes tópicos: *odometry*, *proximity*, *cmd_motors*, *velocity*, *location*, *time_to_impact* e *mapper_activation*. O processo *Collision_Avoidance* recebe

Tabela 5 – Classes de equivalência para os cenários

Cenário	Id/Classe Válida	Id/Classe Inválida
I	$CV_1 : (9 \leq P \leq 127)$	$CI_1 : (P < 9, P > 127)$
II	$CV_2 : (P < 9)$	$CI_2 : (P \geq 9 \text{ e } P \leq 127)$
III	$CV_3 : (P > 127)$	$CI_3 : (P \leq 127)$
IV	$CV_4 : (P \geq 9, P = 6)$	$CI_4 : (P < 9, P \neq 6)$
V	$CV_5 : (P \geq 9, P = 6, P < 4)$	$CI_5 : (P < 9, P \neq 6, P \geq 3)$
VI	$CV_6 : (P \geq 9, P = 6, P \geq 9)$	$CI_6 : (P < 9, P \neq 6, P < 9)$
VII	$CV_7 : (P \geq 9, P = 6)$	$CI_7 : (P < 19, P \neq 6)$
VIII	$CV_8 : (P \geq 19, P = 6, P \geq 9, P < 6)$	$CI_8 : (P < 9, P \neq 6, P < 9, P \geq 6)$
IX	$CV_9 : (P \geq 9, P = 6, P \geq 9, P < 4)$	$CI_9 : (P < 9, P \neq 6, P < 9, P \geq 4)$
X	$CV_{10} : (P \geq 9, P < 4)$	$CI_{10} : (P < 9, P \geq 4)$

Fonte: Elaborada pelo autor.

mensagens por meio dos tópicos *velocity* e *proximity*. Quando um evento relacionado ao tópico *proximity* acontece o método *proximityCallback()* (linha 5 do código) é ativado. Quando o processo *Collision_Avoidance* recebe uma mensagem por meio do tópico *velocity* o método *velocityCallback* é ativado (linha 10 do código), essa mensagem contém um dado de velocidade que é utilizado para calcular o tempo de impacto do robô com um obstáculo. Esse tempo de impacto altera o estado do robô para evitar uma colisão e é publicado no tópico *time_to_impact*, quando o método *publish* (linha 53 do código) é chamado.

O processo *Controller* por sua vez, recebe dados por meio do tópico *time_to_impact* no método *timeToImpactCallback* (linha 70 do código). Essa mensagem contém a variável *time_impactT*, que é utilizada para processar dados sobre os motores, controlar a velocidade e estados do robô e mapear o ambiente. No final do processamento, esse processo publica três mensagens: uma mensagem no tópico *velocity* (linha 135 do código), uma mensagem no tópico *cmd_motors* (linha 136 do código) e outra mensagem no tópico *location* (linha 137 do código). O processo *Controller* recebe também por meio do tópico *odometry* uma mensagem de odometria

Tabela 6 – Conjunto de casos de teste para o sistema do robô aspirador de pó

CT	Classe	Entrada	Saída
1	CV_3, CI_1	(128,129,130,131)	robô exibe mensagem de erro
2	$CV_2, CI_1, CI_4 \dots CI_6, CI_8 \dots CI_{10}$	(8,7,6,5)	robô exibe mensagem de erro
3	CV_1, CI_2, CI_3	(127,126,125,124,123,122)	robô se desloca com velocidade constante
4	CV_9	(12,11,10,6,10,9,3,2,1)	robô se desloca com velocidade constante, reduzida, constante e colide
5	CV_{10}	(9,9,9,3,2,1)	robô se desloca com velocidade constante e colide
6	CI_7	(9,9,9,9,-1,9)	robô se desloca com velocidade constante e exibe mensagem de erro
7	CV_4	(9,9,6,6,6,6,6,6,6)	robô se desloca com velocidade reduzida
8	CV_5	(21,21,21,21,21,21,21,21,6,1)	robô se desloca com velocidade constante, reduzida e para
9	CV_7	(19,18,17,16,15,26,27,28,27,6)	robô se desloca com velocidade constante e reduzida
10	CV_6, CV_8	(9,9,8,7,6,18,17,16,15,4,3)	robô se desloca com velocidade reduzida, constante e colide
11	CV_5, CV_{10}	(9,9,8,7,6,1,18,19,20,21,22)	robô se desloca com velocidade reduzida, para e prossegue com velocidade constante
12	CV_5	(9,9,6,3,3,2,1)	robô colide com obstáculo

Fonte: Elaborada pelo autor.

por meio do método `odometryCallback` (linha 65 do código). A odometria é utilizada para que o robô tenha conhecimento do espaço percorrido no ambiente. Com base no grafo de comunicação e nos elementos do código, o *PSCFG* é gerado (Figura 11).

Figura 11 – *PSCFG* para o sistema do robô aspirador de pó

Fonte: Elaborada pelo autor.

No *PSCFG* os processos são representados por retângulos, interligados por arestas interprocessos (ou tópicos no grafo de comunicação do ROS). Cada retângulo é composto por um ou mais *CFGs* que representam os métodos do processo. As linhas sólidas representam as arestas *intramétodos* ($E^{p,m}$) e as tracejadas representam as arestas *interprocessos* (E_i). Como exemplo, a aresta $(n_3^{4,3}, n_1^{2,1})$ compreende um produtor e um assinante, representando a comunicação entre os processos p_4 (*Controller*) e p_2 (*Collision_Avoidance*). A Tabela 7 contém os conjuntos obtidos a partir das definições no grafo que são utilizadas para a geração dos requisitos de teste.

A partir dessas informações são derivados os elementos requeridos para os critérios de teste. Para este exemplo, os elementos requeridos estão descritos na Tabela 8.

Para esse sistema, uma entrada de teste é uma mensagem a partir do processo *Proximity_Sensor* e a saída esperada consiste em uma mensagem enviada pelo processo *Motor_Driver* para os atuadores do robô (rodas e aspirador), fazendo com que o mesmo se desloque recolhendo partículas do ambiente, reduza a velocidade para evitar colisões ou pare na posição atual. Devido à natureza distribuída do sistema, em que os processos executam em paralelo, as entradas são sequências de mensagens. Conjuntos de teste foram gerados com o objetivo de cobrir os elementos requeridos, gerados pelos critérios (Tabela 8). Para isso, foi utilizado o conjunto de teste previamente gerado com base nos cenários de teste (Tabela 6). Para este exemplo, cujo objetivo é ilustrar a aplicação do modelo de teste foi utilizado o ROS para executar o sistema e

Tabela 7 – Informações geradas pelo modelo de teste para o programa aspirador de pó

$n_p = 5, p_1 = \{m^{1,1}\}, p_2 = \{m^{2,1}, m^{2,2}, m^{2,3}\}, p_3 = \{m^{3,1}, m^{3,2}\}, p_4 = \{m^{4,1}, m^{4,2}, m^{4,3}\},$ $p_5 = \{m^{5,1}, m^{5,2}\}, Prog = \{p_1, p_2, p_3, p_4, p_5\}$
$N^{1,1} = \{n_1^{1,1} \dots n_5^{1,1}\}, N^{2,1} = \{n_1^{2,1}, \dots, n_{11}^{2,1}\}, N^{2,2} = \{n_1^{2,2}\}, N^{2,3} = \{n_1^{2,3} \dots n_4^{2,3}\}, N^{3,1} = \{n_1^{3,1}\},$ $N^{3,2} = \{n_1^{3,2}\}, N^{4,1} = \{n_1^{4,1} \dots n_9^{4,1}\}, N^{4,2} = \{n_1^{4,2}\}, N^{4,3} = \{n_1^{4,3} \dots n_4^{4,3}\},$ $N^{5,1} = \{n_1^{5,1} \dots n_{11}^{5,1}\}, N^{5,2} = \{n_1^{5,2} \dots n_6^{5,2}\},$ $N^1 = N^{1,1}, N^2 = N^{2,1} \cup N^{2,2} \cup N^{2,3}, N^3 = N^{3,1} \cup N^{3,2}, N^4 = N^{4,1} \cup N^{4,2} \cup N^{4,3}, N^5 = N^{5,1} \cup N^{5,2}$ $N = N^1 \cup N^2 \cup N^3 \cup N^4 \cup N^5$
$V^1 = (v1, 'msgProximity'), V^2 = (v2, 'proximityP'), (v3, 'speedP'), V^3 = (v4, 'locationP'),$ $(v5, 'mapP'), V^4 = (v6, 'odometryP'), (v7, 'time_impactP'), V^5 = (v8, 'cmdP')$

Fonte: Elaborada pelo autor.

as entradas que seriam fornecidas pelo sensor em ambiente real ou simulador foram fornecidas pelo testador.

Tabela 8 – Elementos requeridos pelos critérios para o programa aspirador de pó

Critério	Num	Elementos requeridos
todos-nós-comunicação	11	$(n_3^{1,1}), (n_1^{2,2}), (n_1^{2,1}), (n_3^{2,3}), (n_1^{3,1}), (n_1^{3,2}), (n_1^{4,1}), (n_3^{4,3}),$ $(n_1^{4,2}), (n_1^{5,1}), (n_4^{5,2})$
todos-nós-publishers	4	$(n_3^{1,1}), (n_3^{2,3}), (n_3^{4,3}), (n_4^{5,2})$
todos-nós-subscribers	7	$(n_1^{2,2}), (n_1^{2,1}), (n_1^{3,1}), (n_1^{3,2}), (n_1^{4,1}), (n_1^{4,2}), (n_1^{5,1})$
todos-pares-publishers-subscribers	7	$(n_3^{1,1}, n_1^{2,2}), (n_3^{4,3}, n_1^{2,1}), (n_3^{4,3}, n_1^{5,1}), (n_3^{4,3}, n_1^{3,2}), (n_3^{2,3}, n_1^{4,1}),$ $(n_3^{2,3}, n_1^{3,1}), (n_4^{5,2}, n_1^{4,2})$
todos-k-pares-publishers-subscribers	11	$(n_3^{4,3}, n_1^{2,1}, n_3^{2,3}, n_1^{4,1}), (n_3^{4,3}, n_1^{2,1}, n_3^{2,3}, n_1^{3,1}), (n_3^{1,1}, n_1^{2,2}, n_3^{2,3},$ $n_1^{4,1}), (n_3^{1,1}, n_1^{2,2}, n_3^{2,3}, n_1^{3,1}), (n_4^{5,2}, n_1^{4,2}, n_3^{4,3}, n_1^{2,1}), (n_4^{5,2}, n_1^{4,2},$ $n_3^{4,3}, n_1^{5,1}), (n_3^{4,3}, n_1^{5,1}, n_4^{5,2}, n_1^{4,2}), (n_3^{2,3}, n_1^{4,1}, n_3^{4,3}, n_1^{2,1}), (n_3^{2,3},$ $n_1^{4,1}, n_3^{4,3}, n_1^{3,2}), (n_3^{2,3}, n_1^{4,1}, n_3^{4,3}, n_1^{5,1}), (n_4^{5,2}, n_1^{4,2}, n_3^{4,3}, n_1^{3,2})$
todas-k-mensagens	14	$(n_1^{2,2}, proximityP_1, \dots, proximityP_{10}),$ $(n_1^{2,2}, proximityP_{11}, \dots, proximityP_{20})$ $(n_1^{2,1}, speedP_1, \dots, speedP_{10}),$ $(n_1^{2,1}, speedP_{11}, \dots, speedP_{20}),$ $(n_1^{3,1}, mapP_1, \dots, mapP_{10}),$ $(n_1^{3,1}, mapP_{11}, \dots, mapP_{20}),$ $(n_1^{3,2}, locationP_1, \dots, locationP_{10}),$ $(n_1^{3,2}, locationP_{11}, \dots, locationP_{20}),$ $(n_1^{4,1}, time_impactP_1, \dots, time_impactP_{10}),$ $(n_1^{4,1}, time_impactP_{11}, \dots, time_impactP_{20})$ $(n_1^{4,2}, odometryP_1, \dots, odometryP_{10}),$ $(n_1^{4,2}, odometryP_{11}, \dots, odometryP_{20}),$ $(n_1^{5,1}, cmdP_1, \dots, cmdP_{10})$ $(n_1^{5,1}, cmdP_{11}, \dots, cmdP_{20})$

Fonte: Elaborada pelo autor.

Durante a execução dos testes os cenários e elementos requeridos pelos critérios são avaliados quanto à cobertura. Todos os cenários de teste especificados no Quadro 2 foram satisfeitos. As entradas percorrem um conjunto de caminhos, e com o conhecimento destes caminhos identifica-se as coberturas para os elementos requeridos. Para o conjunto de casos de teste apresentado na Tabela 6 as coberturas dos elementos requeridos foram 100% para todos os critérios.

As coberturas máximas obtidas devem-se às características do sistema. Neste sistema todos os tópicos registrados no processo mestre são exercitados durante a execução, visto que só é utilizada comunicação baseada em tópicos. Para o critério todos-k-pares-publishers-subscribers os pares de arestas $(n_3^{4,3}, n_1^{2,1}, n_3^{2,3}, n_1^{4,1})$ e $(n_3^{4,3}, n_1^{2,1}, n_3^{2,3}, n_1^{3,1})$ são não executáveis, considerando a mesma execução do processo para uma entrada em específico. Considerando uma determinada execução, o tópico $(n_3^{2,3}, n_1^{4,1})$ é ativado antes do tópico $(n_3^{4,3}, n_1^{2,1})$, de forma que a ordem de execução imposta para o elemento requerido é não executável. O mesmo ocorre para o outro elemento requerido. Para o critério todas-k-mensagens, a cobertura está diretamente relacionada com as mensagens enviadas e recebidas. Na inicialização do sistema as primeiras mensagens são perdidas até que os processos estejam em execução e trocando dados, situação exercitada com os casos de teste 4, 5 e 12. Essa característica provoca a sobrescrita de mensagens, caso as frequências dos processos comunicantes não seja a ideal. O critério todas-k-mensagens obteve neste caso a cobertura máxima possível, dado que não foram cobertos somente os elementos requeridos, cujas primeiras mensagens são perdidas na inicialização do sistema. Como este critério verifica a ordem e o conteúdo das mensagens, o envio sem o correspondente recebimento é considerado não coberto e para este caso, 8 elementos requeridos não foram cobertos pelo conjunto de teste, porém são elementos não executáveis.

Além da avaliação de cobertura para os critérios de teste, para avaliar a eficácia do conjunto de teste, defeitos foram inseridos no sistema do robô aspirador de pó, gerando versões do sistema com defeitos. Os defeitos desse domínio estão relacionados na Seção 5.3 e destes, três relacionados com a comunicação entre os processos foram selecionados para semeadura de defeitos:

- (i) Alteração na frequência de publicação das mensagens, que ocorre quando um processo envia dados mais rápido do que a capacidade de recebimento do receptor, ou o contrário;
- (ii) Alteração no tamanho da fila de um *subscriber* que recebe dados a partir de diferentes tópicos;
- (iii) Retirada de um comando de publicação de dados em um processo.

Os defeitos dos tipos *i* e *ii* estão relacionados em Carlson, Murphy e Nelson (2004) como falhas de interação ou enganos cometidos pelo programador durante o desenvolvimento. O defeito do tipo *iii* está relacionado com uma falha em um sensor, ou outro módulo que deixa

de enviar um dado esperado por outro processo. Para cada defeito foram geradas 3 versões do sistema, assim, 9 versões foram utilizadas para a avaliação do conjunto de casos de teste.

No primeiro programa, Def-i-1, o defeito inserido alterou a frequência do processo *Proximity* de 0.6 para 0.9, que passou a enviar dados de forma mais rápida e, assim, o processo *Collision_Avoidance* deixou de receber algumas mensagens que foram sobrescritas no *buffer* do *subscriber*. No segundo programa, Def-i-2, a frequência de publicação do processo *Collision_Avoidance* foi alterada de 0.6 para 0.2, e no terceiro programa, Def-i-3, a frequência de publicação do processo *Controller* foi alterada de 0.6 para 0.3. No quarto programa, Def-ii-1, a fila do método *velocityCallback* no processo *Collision_Avoidance* foi alterada de 1 para o tamanho 10. No programa 5, Def-ii-2, a fila do método *timeToImpactCallback* do processo *Controller* foi alterada de 1 para 100, e, no sexto programa, Def-ii-3, a fila do método *cmdCallback* do processo *Motor_Driver* foi alterada de 1 para 1000. Aqui, dificilmente ocorreriam problemas durante a execução do programa, pois normalmente os problemas ocorrem quando está sendo utilizado um tamanho de *buffer* menor que o esperado, porém, neste exemplo o programa já especificava a utilização de filas de tamanho 1 e essa característica foi assim mantida. No sétimo programa, Def-iii-1, foi retirado o comando de publicação da mensagem *msgTimeToImpact* pelo processo *Collision_Avoidance*, o oitavo o programa, Def-iii-2, não continha o comando de publicação da mensagem *msgVelocity* pelo processo *Controller* e, no nono programa, Def-iii-3, foi retirado o comando de publicação da mensagem *msgProximity* do processo *Proximity*.

Durante a execução não foram usados *drivers* ou *stubs*, somente marcações no código para verificação da cobertura. O Quadro 3 mostram os resultados das execuções dos programas com defeitos.

Quadro 3 – Resultados das execuções dos programas com defeitos

Versões com defeitos	Casos de teste											
	1	2	3	4	5	6	7	8	9	10	11	12
Def-i-1	N	N	N	N	N	N	N	N	N	N	N	S
Def-i-2	N	N	N	N	N	N	N	S	N	N	N	N
Def-i-3	N	N	N	N	N	N	N	S	N	N	N	S
Def-ii-1	N	N	N	N	N	N	N	N	N	N	N	S
Def-ii-2	N	N	N	N	N	N	N	N	N	N	N	S
Def-ii-3	N	N	N	N	N	N	N	N	N	N	N	S
Def-iii-1	N	N	S	N	S	S	S	S	S	S	S	S
Def-iii-2	N	N	S	N	S	S	S	S	S	S	S	S
Def-iii-3	N	N	N	N	S	S	S	S	S	S	S	S

Fonte: Elaborada pelo autor.

A primeira coluna refere-se aos identificadores das versões dos programas com defeitos e as colunas seguintes mostram os resultados da execução para cada dado de entrada do conjunto de teste. *S* indica que o caso de teste revelou o defeito, portanto foi obtido para esse caso de teste uma saída não esperada, e *N* indica que o caso de teste não revelou o defeito e assim, foi obtido para esse dado de teste uma saída esperada. De acordo com o Quadro 3, o primeiro, o segundo e

o quarto caso de teste não revelaram nenhum dos defeitos, porém, os demais casos de teste do conjunto foram capazes de revelar um dos defeitos, de modo que, no conjunto, todos os defeitos foram revelados.

Dessa forma, a abordagem de teste aqui apresentada explora erros típicos em sistemas robóticos, tais como:

- Mensagens perdidas ou sobrescritas quando são utilizadas frequências incorretas entre processos comunicantes;
- Mensagens perdidas quando o tempo de processamento dos *callbacks* não é o ideal;
- Mensagens duplicadas e corrompidas;
- Mensagens atrasadas;

Com o intuito de avaliar essas situações o modelo de teste proposto mostrou-se eficaz em revelar defeitos típicos, além de apoiar o testador no controle da atividade de teste, permitindo explorar mais detalhadamente características de comunicação entre os processos, contribuindo para uma maior sistematização da atividade de teste de sistemas robóticos móveis.

5.6 Considerações finais

Este capítulo apresentou a abordagem de teste de integração proposta para exercitar características específicas do modelo de interação *publish/subscribe* relacionadas à comunicação por tópicos entre os processos, como não determinismo das entradas e sincronizações entre os processos que não são totalmente cobertos pelas abordagens de teste existentes. A abordagem de teste é composta por cenários, gerados com base em teste funcional e uma família de critérios de cobertura, baseada em teste estrutural, que considera o fluxo de controle e a comunicação entre os processos. Neste capítulo também foi apresentado um exemplo de uso da abordagem utilizando um sistema cuja especificação foi extraída da literatura. Com o exemplo de uso, a abordagem de teste mostrou-se eficaz em exercitar a comunicação entre os processos do sistema e em revelar defeitos típicos desse domínio. O próximo capítulo apresenta dois estudos de caso exploratórios conduzidos para a avaliação da abordagem de teste proposta.

AVALIAÇÃO DA ABORDAGEM DE TESTE DE INTEGRAÇÃO PARA SISTEMAS ROBÓTICOS MÓVEIS

6.1 Considerações iniciais

Este capítulo relata os estudos de caso exploratórios conduzidos com o objetivo de avaliar a aplicabilidade da abordagem de teste proposta nesta tese de doutorado. Foram conduzidos dois estudos utilizando sistemas mais complexos que o exemplo de uso apresentado no capítulo anterior. As avaliações tentam cobrir dois aspectos: i) adequação ao lidar com sistemas mais complexos e; ii) aplicabilidade da abordagem de teste. Os estudos tiveram como participante a própria pesquisadora. Os objetos foram aplicações do domínio de sistemas robóticos móveis desenvolvidas por alunos do ICMC/USP. O primeiro estudo de caso exploratório fez uso de um sistema robótico desenvolvido por alunos do Laboratório de Robótica Móvel (LRM). Em seguida, foi realizado um segundo estudo de caso exploratório utilizando um sistema robótico desenvolvido por alunos de iniciação científica e de doutorado do Laboratório de Engenharia de Software (LabES). O capítulo está organizado da seguinte forma. Na Seção 6.2 é apresentado o estudo de caso exploratório para o primeiro sistema. Na Seção 6.3 é apresentado o estudo de caso exploratório para o segundo sistema. Na Seção 6.4 é apresentada uma análise teórica sobre os tipos de defeitos revelados pela abordagem. Na Seção 6.5 é apresentado um estudo comparativo entre a abordagem proposta e outros estudos da literatura. Na Seção 6.6 estão as considerações finais do capítulo.

6.2 Estudo de caso exploratório I: Carina Parking

Neste primeiro estudo a abordagem de teste proposta nesta tese foi explorada com o objetivo de verificar sua aplicabilidade. A abordagem tem sua aplicabilidade indicada caso os resultados apontem um custo de aplicação viável e contribuições quanto a capacidade de revelar defeitos no sistema utilizado no estudo.

Para este primeiro estudo foi utilizado o Carina Parking (MATIAS *et al.*, 2015), que é um sistema de estacionamento projetado como parte do projeto CaRINA¹. Neste sistema, em ambiente de simulação, o usuário fornece um objetivo (ponto no plano e sentido) e o robô se desloca do ponto inicial até o ponto objetivo. O objetivo representa uma vaga em que o robô deve estacionar.

Para que o objetivo seja alcançado, o robô deve planejar, atualizar e seguir a rota calculada evitando obstáculos do ponto inicial ao objetivo. O Carina Parking foi desenvolvido utilizando o ROS e o simulador MORSE². A Figura 12 ilustra o sistema em ambiente de simulação. A janela apresentada na figura é da *Rviz*³, que é uma ferramenta de visualização tridimensional integrada ao ROS. A *Rviz* permite ao usuário interagir com o simulador, selecionando opções como: 1) setar pontos inicial e objetivo para as rotas, utilizando o menu principal; 2) alterar configurações da simulação, por meio das opções do lado esquerdo da janela; 3) alterar a forma de visualizar a simulação, usando as opções do lado direito da janela, dentre outras. A *Rviz* também permite visualizar o que está sendo simulado em mais detalhes, como exibir os eixos de rotação em relação à base do carro robótico.

A Figura 13 apresenta o grafo de comunicação gerado pelo ROS no momento da simulação. Observa-se por meio da figura, que o Carina Parking é composto por 10 processos:

- Os processos *link1*, *link2* e *link3* enviam dados (transformações) de odometria e sensores;
- *Slam_gmapping* gera o mapa usando dados a partir dos sensores;
- O processo *map_server* salva o mapa e envia-o para outros nós que solicitarem;
- O processo *morse* atua como intermediário entre o simulador e o ROS, fornecendo dados que seriam obtidos pelos sensores em um ambiente real;
- *Amcl* calcula a localização do robô no ambiente;
- *Move_base* recebe dados dos sensores, transformações e mapa e planeja a trajetória enviando comandos de velocidade para o robô;

¹ <http://irm.icmc.usp.br/web/index.php?n=Port.ProjCarina2Info>

² <https://www.openrobots.org/wiki/morse/>

³ <http://wiki.ros.org/rviz>

- *Ackermann_pub* recebe comandos de velocidade do nó *move_base* e converte-os considerando as características do robô, como por exemplo, este robô contém 4 rodas ao invés de 2;
- *Ackermann_sub* age como um terminal para exibir comandos de velocidade;
- *Rviz*, que faz a interface entre a *Rviz* e o sistema.

Figura 12 – Carina Parking em ambiente de simulação

Fonte: Elaborada pelo autor.

Figura 13 – Grafo de comunicação para o Carina Parking

Fonte: Elaborada pelo autor.

Esta versão do Carina Parking possui como limitações casos em que o robô alcança o objetivo mas não segue o sentido requerido pelo usuário. Isso ocorre, pois o robô possui apenas um sensor laser na parte dianteira que permite obter somente percepção frontal. Como o robô

não tem percepção traseira, o comportamento esperado é que ele manobre o carro verificando se o local é seguro para estacionar.

6.2.1 Execução do teste - Carina Parking

Para a aplicação da abordagem de teste de integração o primeiro passo é a geração dos cenários para o sistema em teste com base na especificação. Os cenários gerados são apresentados no Quadro 4.

Quadro 4 – Cenários para o Carina Parking

Cenário	Descrição do cenário
I	Verificar se o robô estaciona no objetivo sem dados de odometria
II	Verificar se o robô estaciona no objetivo sem dados do sensor laser
III	Verificar se o robô estaciona no objetivo com dados do sensor laser não esperados
IV	Verificar se o robô estaciona no objetivo quando o mesmo é definido fora do estacionamento
V	Verificar se o robô estaciona no objetivo quando o mesmo é definido em um obstáculo
VI	Verificar se o robô estaciona em vaga à frente do robô e à direita
VII	Verificar se o robô estaciona em vaga à frente do robô e à esquerda
VIII	Verificar se o robô estaciona em vaga atrás do robô e à direita
IX	Verificar se o robô estaciona em vaga atrás do robô e à esquerda
X	Verificar se o robô estaciona em vaga com sentidos diferentes da esquerda para a direita e vice-versa

Fonte: Elaborada pelo autor.

Para esse sistema, as entradas são: 1) ponto objetivo no plano (P); 2) sentido do vetor objetivo (S); 3) dados do sensor laser (L) e; 4) dados de odometria (O). Para a geração dos dados de teste é utilizado o critério funcional particionamento de equivalência, no qual, para cada cenário são definidas as classes válidas e inválidas (Tabela 9) com base na especificação. Para esse sistema há 4 entradas, e, portanto, as classes referem-se aos cenários e entradas que estabeleçam aquele cenário. Como exemplo, o cenário *I* que deseja verificar se o robô estaciona no objetivo sem dados de odometria. Para esse caso, apenas a entrada referente à odometria varia para as classes válidas e inválidas.

Além das entradas, com base na especificação as seguintes saídas podem ser obtidas: 1) robô estaciona no objetivo; 2) robô não estaciona no objetivo; 3) robô estaciona no objetivo com sentido contrário e; 4) objetivo inválido.

Tabela 9 – Classes de equivalência para os cenários do Carina Parking

Cenário	Entrada	Id/Classe válida	Id/Classe inválida
I	P	$CV_1 : P \in \text{Plano e } P \not\subset \text{Obstáculo}$	$CI_1 : P \in \text{Plano e } P \not\subset \text{Obstáculo}$
	S	esquerda p/ direita e direita p/ esquerda	esquerda p/ direita e direita p/ esquerda
	L	dados válidos	dados válidos
	O	dados inválidos	dados válidos
II	P	$CV_2 : P \in \text{Plano e } P \not\subset \text{Obstáculo}$	$CI_2 : P \in \text{Plano e } P \not\subset \text{Obstáculos}$
	S	esquerda p/ direita e direita p/ esquerda	esquerda p/ direita e direita p/ esquerda
	L	dados inválidos ou $\emptyset$	dados válidos
	O	dados válidos	dados válidos
III	P	$CV_3 : P \in \text{Plano e } P \not\subset \text{Obstáculo}$	$CI_3 : P \in \text{Plano e } P \not\subset \text{Obstáculo}$
	S	esquerda p/ direita e direita p/ esquerda	esquerda p/ direita e direita p/ esquerda
	L	dados com frequência e range não esperados	dados válidos, inválidos ou $\emptyset$
	O	dados válidos	dados válidos
IV	P	$CV_4 : P \notin \text{Plano}$	$CI_4 : P \in \text{Plano}$
	S	esquerda p/ direita e direita p/ esquerda	esquerda p/ direita e direita p/ esquerda
	L	dados válidos	dados válidos
	O	dados válidos	dados válidos
V	P	$CV_5 : P \subset \text{Obstáculo}$	$CI_5 : P \not\subset \text{Obstáculo}$
	S	esquerda p/ direita e direita p/ esquerda	esquerda p/ direita e direita p/ esquerda
	L	dados de laser válidos	dados válidos
	O	dados de odometria válidos	dados válidos
VI-IX	P	$CV_6 : P \in \text{Plano e } P \not\subset \text{Obstáculo}$	$CI_6 : P \notin \text{Plano e } P \subset \text{Obstáculo}$
	S	esquerda p/ direita e direita p/ esquerda	esquerda p/ direita e direita p/ esquerda
	L	dados válidos	dados válidos
	O	dados válidos	dados válidos
X	P	$CV_7 : P \in \text{Plano e } P \not\subset \text{Obstáculo}$	$CI_7 : P \in \text{Plano e } P \not\subset \text{Obstáculos}$
	S	sentidos intermediários	esquerda p/ direita e direita p/ esquerda
	L	dados válidos	dados válidos
	O	dados válidos	dados válidos

Fonte: Elaborada pelo autor.

Nota: P - proximidade, S - sentido, L - dados do sensor laser, O - dados de odometria

A partir das classes de equivalência e saídas são gerados os casos de teste (Tabela 10) para satisfazer os cenários de teste (Quadro 4). Os casos de teste foram gerados com base nas informações fornecidas pelo simulador sobre os pontos inicial e objetivo visualizados usando a ferramenta *Rviz*. Um cenário pode ter mais de um caso de teste para exercitar todas as possibilidades de um determinado cenário. Para o cenário *III*, por exemplo, o que diferencia os dois casos de teste são as configurações do laser, no caso de teste 3, o laser possui frequência 10 e alcance 30, no caso de teste 4, a frequência é 10 e o alcance 0. Como outro exemplo, no cenário *V*

os dois casos de teste avaliam diferentes pontos objetivos inválidos do mapa de estacionamento.

Tabela 10 – Conjunto de casos de teste para o Carina Parking

Código		Entradas			Saída	
C	CE	CT	Ponto inic., ponto obj.	L	O	
I	CV ₁	1	(442, 434, 0), (492, 385, 0)	v	i	Robô não estaciona no objetivo
II	CV ₂	2	(442, 434, 0), (492, 385, 0)	i	v	Robô estaciona no objetivo
III	CV ₃	3	(442, 434, 0), (492, 385, 0)	n	v	Robô estaciona no objetivo
	CV ₃	4	(442, 434, 0), (492, 385, 0)	n	v	Robô estaciona no objetivo
IV	CV ₄ , CI ₆	5	(442, 434, 0), (492, 643, 0)	v	v	Objetivo inválido
V	CV ₅ , CI ₆	6	(442, 434, 0), (320, 530, 14)	v	v	Objetivo inválido
	CV ₅ , CI ₆	7	(442, 434, 0), (356, 527, 0)	v	v	Objetivo inválido
VI	CV ₆ , CI ₁ ...CI ₅ , CI ₇	8	(442, 434, 0), (496, 528, 0)	v	v	Robô estaciona no objetivo
	CV ₆ , CI ₁ ...CI ₅ , CI ₇	9	(442, 434, 0), (530, 540, 8)	v	v	Robô estaciona no objetivo
	CV ₆ , CI ₁ ...CI ₅ , CI ₇	10	(442, 434, 0), (539, 533, 8)	v	v	Robô estaciona no objetivo
VII	CV ₆ , CI ₁ ...CI ₅ , CI ₇	11	(142, 434, 0), (364, 565, 8)	v	v	Robô estaciona no objetivo
	CV ₆ , CI ₁ ...CI ₅ , CI ₇	12	(442, 434, 0), (335, 584, 0)	v	v	Robô estaciona no objetivo
	CV ₆ , CI ₁ ...CI ₅ , CI ₇	13	(442, 434, 0), (294, 602, 8)	v	v	Robô estaciona no objetivo
VIII	CV ₆ , CI ₁ ...CI ₅ , CI ₇	14	(442, 434, 0), (543, 264, 0)	v	v	Robô estaciona no objetivo
	CV ₆ , CI ₁ ...CI ₅ , CI ₇	15	(442, 434, 0), (552, 263, 8)	v	v	Robô estaciona no objetivo
	CV ₆ , CI ₁ ...CI ₅ , CI ₇	16	(442, 434, 0), (515, 387, 0)	v	v	Robô estaciona no objetivo
	CV ₆ , CI ₁ ...CI ₅ , CI ₇	17	(442, 434, 0), (517, 398, 8)	v	v	Robô estaciona no objetivo com sentido contrário
IX	CV ₆ , CI ₁ ...CI ₅ , CI ₇	18	(442, 434, 4), (365, 348, 0)	v	v	Robô estaciona no objetivo
	CV ₆ , CI ₁ ...CI ₅ , CI ₇	19	(442, 434, 0), (375, 352, 8)	v	v	Robô estaciona no objetivo
	CV ₆ , CI ₁ ...CI ₅ , CI ₇	20	(442, 434, 0), (335, 258, 0)	v	v	Robô estaciona no objetivo
	CV ₆ , CI ₁ ...CI ₅ , CI ₇	21	(442, 434, 0), (343, 256, 8)	v	v	Robô estaciona no objetivo
	CV ₆ , CI ₁ ...CI ₅ , CI ₇	22	(442, 434, 0), (313, 298, 8)	v	v	Robô estaciona no objetivo
	CV ₆ , CI ₁ ...CI ₅ , CI ₇	23	(442, 434, 0), (319, 297, 0)	v	v	Robô estaciona no objetivo
X	CV ₇ , CI ₁	24	(442, 434, 0), (379, 485, 6)	v	v	Objetivo inválido
	CV ₇ , CI ₁	25	(442, 434, 0), (347, 578, 14)	v	v	Objetivo inválido
	CV ₇ , CI ₁	26	(442, 434, 0), (517, 398, 2)	v	v	Objetivo inválido
	CV ₇ , CI ₁	27	(442, 434, 0), (520, 339, 9)	v	v	Objetivo inválido

Fonte: Elaborada pelo autor.

Nota: C - cenário de teste, CE - classe de equivalência, CT - caso de teste, L - dados do sensor laser, O - dados de odometria, n - dados fora da faixa de valores esperados, i - ausência de dado, v - dado dentro da faixa de valores esperados

Após a geração dos cenários, a abordagem prevê a aplicação dos critérios de teste com

base na comunicação entre os processos do sistema. Para isso, o PSCFG é gerado a partir do grafo de comunicação (Figura 13) e elementos do código fonte. A Figura 14 ilustra o PSCFG para o Carina Parking, usado para a geração dos elementos requeridos pelos critérios.

Para a construção do PSCFG foram desconsideradas características de comunicação por serviços e *actions* que não são o foco de estudo desta abordagem de teste. Também foram desconsiderados tópicos que não são utilizados (nenhum processo publica ou subscreve dados). A Tabela 11 contém os conjuntos obtidos a partir das definições no PSCFG que são utilizados para a geração dos elementos requeridos.

Tabela 11 – Informações geradas pelo modelo de teste para o Carina Parking

$n_p = 10, p_1 = \{m^{1,1}, m^{1,2}, m^{1,3}, m^{1,4}\}, p_2 = \{m^{2,1}, m^{2,2}, m^{2,3}, m^{2,4}\}, p_3 = \{m^{3,1}, m^{3,2}, m^{3,3}, m^{3,4}\},$ $p_4 = \{m^{4,1}, m^{4,2}, m^{4,3}, m^{4,4}, m^{4,5}, m^{4,6}, m^{4,7}, m^{4,8}\}, p_5 = \{m^{5,1}\}, p_6 = \{m^{6,1}, m^{6,2}, m^{6,3}\},$ $p_7 = \{m^{7,1}, m^{7,2}, m^{7,3}, m^{7,4}, m^{7,5}, m^{7,6}, m^{7,7}, m^{7,8}\}, p_8 = \{m^{8,1}, m^{8,2}, m^{8,3}\}, p_9 = \{m^{9,1}\},$ $p_{10} = \{m^{10,1}\},$ $Prog = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\}$
$N^{1,1} = \{n_1^{1,1}\}, N^{1,2} = \{n_1^{1,2}\}, N^{1,3} = \{n_1^{1,3}\}, N^{1,4} = \{n_4^{1,4}\}, N^{2,1} = \{n_1^{2,1}\}, N^{2,2} = \{n_1^{2,2}\},$ $N^{2,3} = \{n_1^{2,3}\}, N^{2,4} = \{n_4^{2,4}\}, N^{3,1} = \{n_1^{3,1}\}, N^{3,2} = \{n_1^{3,2}\}, N^{3,3} = \{n_1^{3,3}\}, N^{3,4} = \{n_4^{3,4}\},$ $N^{4,1} = \{n_{67}^{4,1}\}, N^{4,2} = \{n_7^{4,2}\}, N^{4,3} = \{n_4^{4,3}\}, N^{4,4} = \{n_1^{4,4}\}, N^{4,5} = \{n_1^{4,5}\}, N^{4,6} = \{n_1^{4,6}\},$ $N^{4,7} = \{n_4^{4,7}\}, N^{4,8} = \{n_{27}^{4,8}\}, N^{5,1} = \{n_{20}^{5,1}\}, N^{6,1} = \{n_1^{6,1}\}, N^{6,2} = \{n_1^{6,2}\}, N^{6,3} = \{n_1^{6,3}\},$ $N^{7,1} = \{n_{29}^{7,1}\}, N^{7,2} = \{n_1^{7,2}\}, N^{7,3} = \{n_1^{7,3}\}, N^{7,4} = \{n_3^{7,4}, \dots, n_{59}^{7,4}\}, N^{7,5} = \{n_1^{7,5}\}, N^{7,6} = \{n_1^{7,6}\},$ $N^{7,7} = \{n_4^{7,7}\}, N^{7,8} = \{n_{21}^{7,8}\}, N^{8,1} = \{n_1^{8,1}\}, N^{8,2} = \{n_1^{8,2}\}, N^{8,3} = \{n_{18}^{8,3}\}, N^{9,1} = \{n_1^{9,1}\}, \dots, n_4^{9,1}\}$ $N^{10,1} = \{n_1^{10,1}\},$ $N^1 = N^{1,1} \cup N^{1,2} \cup N^{1,3} \cup N^{1,4}, N^2 = N^{2,1} \cup N^{2,2} \cup N^{2,3} \cup N^{2,4}$ $N^3 = N^{3,1} \cup N^{3,2} \cup N^{3,3} \cup N^{3,4}, N^4 = N^{4,1} \cup N^{4,2} \cup N^{4,3} \cup N^{4,4} \cup N^{4,5} \cup N^{4,6} \cup N^{4,7} \cup N^{4,8}$ $N^5 = N^{5,1}, N^6 = N^{6,1} \cup N^{6,2} \cup N^{6,3}, N^7 = N^{7,1} \cup N^{7,2} \cup N^{7,3} \cup N^{7,4} \cup N^{7,5} \cup N^{7,6} \cup N^{7,7} \cup N^{7,8}$ $N^8 = N^{8,1} \cup N^{8,2} \cup N^{8,3}, N^9 = N^{9,1}, N^{10} = N^{10,1}$ $N = N^1 \cup N^2 \cup N^3 \cup N^4 \cup N^5 \cup N^6 \cup N^7 \cup N^8 \cup N^9 \cup N^{10}$
$V^4 = \{ (v1, 'scan'), (v2, 'tf'), (v3, 'scan') \}$ $V^7 = \{ (v4, 'msg'), (v5, 'msg'), (v6, 'laser_scan'), (v7, 'tf'), (v8, 'tf') \}$ $V^8 = \{ (v9, 'tf') \}$ $V^9 = \{ (v10, 'data') \}$ $V^{10} = \{ (v11, 'data') \}$ $V = V^4 \cup V^7 \cup V^8 \cup V^9 \cup V^{10}$

Fonte: Elaborada pelo autor.

A partir dessas informações e PSCFG são derivados os elementos requeridos para os critérios de teste. Para este sistema os elementos requeridos estão descritos na Tabela 12.

Figura 14 – PSCFG para o Carina Parking

Fonte: Elaborada pelo autor.

Tabela 12 – Elementos requeridos para o Carina Parking

Crítério	Elementos requeridos
Todos-nós-comunicação	$n_1^{9,1}, n_4^{9,1}, n_1^{10,1}, n_{27}^{4,8}, n_7^{4,2}, n_4^{4,3}, n_1^{8,1}, n_1^{8,2}, n_{18}^{8,3}, n_1^{6,1}, n_1^{6,2}, n_1^{6,3}, n_{20}^{5,1}, n_1^{1,1}, n_1^{2,1}, n_1^{3,1}, n_{29}^{7,1},$ $n_1^{7,2}, n_1^{7,3}, n_3^{7,4}, n_{54}^{7,4}, n_{59}^{7,4}, n_{21}^{7,8}, n_1^{11,1}, n_{67}^{4,1}$
Todos-nós-publishers	$n_1^{1,1}, n_1^{2,1}, n_1^{3,1}, n_{27}^{4,8}, n_4^{4,3}, n_{20}^{5,1}, n_1^{6,1}, n_1^{6,2}, n_1^{6,3}, n_{54}^{7,4}, n_{59}^{7,4}, n_1^{8,2}, n_4^{9,1}, n_1^{11,1}, n_{18}^{8,3}$
Todos-nós-subscribers	$n_7^{4,2}, n_1^{7,2}, n_1^{7,3}, n_3^{7,4}, n_{21}^{7,8}, n_{29}^{7,1}, n_1^{8,1}, n_1^{9,1}, n_1^{10,1}, n_{67}^{4,1}$
Todos-pares-publishers-subscribers	$(n_1^{8,2}, n_1^{9,1}), (n_{18}^{8,3}, n_1^{9,1}), (n_4^{9,1}, n_1^{10,1}), (n_{20}^{5,1}, n_1^{8,1}), (n_{20}^{5,1}, n_1^{7,3}), (n_{27}^{4,8}, n_1^{8,1}), (n_{27}^{4,8}, n_1^{7,3}),$ $(n_1^{11,1}, n_1^{7,2}), (n_7^{4,2}, n_1^{8,1}), (n_7^{4,2}, n_3^{7,4}), (n_1^{6,2}, n_1^{8,1}), (n_1^{6,2}, n_3^{7,4}), (n_1^{6,3}, n_1^{8,1}), (n_1^{1,1}, n_1^{8,1}),$ $(n_1^{1,1}, n_{21}^{7,8}), (n_1^{2,1}, n_1^{8,1}), (n_1^{2,1}, n_{21}^{7,8}), (n_1^{3,1}, n_1^{8,1}), (n_1^{3,1}, n_{21}^{7,8}), (n_4^{4,3}, n_1^{8,1}), (n_4^{4,3}, n_{21}^{7,8}),$ $(n_1^{6,3}, n_1^{4,8}), (n_{54}^{7,4}, n_1^{8,1}), (n_{54}^{7,4}, n_{67}^{4,1}), (n_{59}^{7,4}, n_1^{8,1}), (n_{59}^{7,4}, n_{67}^{4,1}), (n_4^{4,3}, n_{29}^{7,8})$
Todos-k-pares-publishers-subscribers	$(n_{20}^{5,1}, n_1^{8,1}, n_{18}^{8,3}, n_1^{9,1}), (n_{20}^{5,1}, n_1^{8,1}, n_1^{8,2}, n_1^{9,1}), (n_1^{8,2}, n_1^{9,1}, n_4^{9,1}, n_1^{10,1}), (n_{18}^{8,3}, n_1^{9,1}, n_4^{9,1}, n_1^{10,1}),$ $(n_1^{1,1}, n_1^{8,1}, n_{18}^{8,3}, n_1^{9,1}), (n_1^{1,1}, n_1^{8,1}, n_1^{8,2}, n_1^{9,1}), (n_1^{2,1}, n_1^{8,1}, n_{18}^{8,3}, n_1^{9,1}), (n_1^{2,1}, n_1^{8,1}, n_1^{8,2}, n_1^{9,1}),$ $(n_1^{3,1}, n_1^{8,1}, n_{18}^{8,3}, n_1^{9,1}), (n_1^{3,1}, n_1^{8,1}, n_1^{8,2}, n_1^{9,1}), (n_1^{6,1}, n_1^{8,1}, n_{18}^{8,3}, n_1^{9,1}), (n_1^{6,1}, n_1^{8,1}, n_1^{8,2}, n_1^{9,1}),$ $(n_1^{6,2}, n_1^{8,1}, n_{18}^{8,3}, n_1^{9,1}), (n_1^{6,2}, n_1^{8,1}, n_1^{8,2}, n_1^{9,1}), (n_1^{6,3}, n_1^{8,1}, n_{18}^{8,3}, n_1^{9,1}), (n_1^{6,3}, n_1^{8,1}, n_1^{8,2}, n_1^{9,1}),$ $(n_{27}^{4,8}, n_1^{8,1}, n_{18}^{8,3}, n_1^{9,1}), (n_{27}^{4,8}, n_1^{8,1}, n_1^{8,2}, n_1^{9,1}), (n_4^{4,3}, n_1^{8,1}, n_{18}^{8,3}, n_1^{9,1}), (n_4^{4,3}, n_1^{8,1}, n_1^{8,2}, n_1^{9,1}),$ $(n_{54}^{7,4}, n_1^{8,1}, n_{18}^{8,3}, n_1^{9,1}), (n_{54}^{7,4}, n_1^{8,1}, n_1^{8,2}, n_1^{9,1}), (n_{59}^{7,4}, n_1^{8,1}, n_{18}^{8,3}, n_1^{9,1}), (n_{59}^{7,4}, n_1^{8,1}, n_1^{8,2}, n_1^{9,1}),$ $(n_{20}^{5,1}, n_1^{7,3}, n_{54}^{7,4}, n_1^{8,1}), (n_{20}^{5,1}, n_1^{7,3}, n_{59}^{7,4}, n_1^{8,1}), (n_1^{11,1}, n_1^{7,2}, n_{54}^{7,4}, n_1^{8,1}), (n_1^{11,1}, n_1^{7,2}, n_{59}^{7,4}, n_1^{8,1}),$ $(n_1^{11,1}, n_1^{7,2}, n_{54}^{7,4}, n_{67}^{4,1}), (n_1^{11,1}, n_1^{7,2}, n_{59}^{7,4}, n_{67}^{4,1}), (n_1^{8,1}, n_3^{7,4}, n_{54}^{7,4}, n_{67}^{4,1}), (n_1^{8,1}, n_3^{7,4}, n_{59}^{7,4}, n_{67}^{4,1}),$ $(n_1^{8,1}, n_7^{4,2}, n_{27}^{4,8}, n_3^{7,4}), (n_1^{8,1}, n_7^{4,2}, n_4^{4,3}, n_{21}^{7,8}), (n_1^{8,1}, n_7^{4,2}, n_4^{4,3}, n_{29}^{7,8}), (n_4^{4,3}, n_{29}^{7,8}, n_{59}^{7,4}, n_1^{8,1}),$ $(n_4^{4,3}, n_{21}^{7,8}, n_{54}^{7,4}, n_1^{8,1}), (n_4^{4,3}, n_{21}^{7,8}, n_{59}^{7,4}, n_1^{8,1}), (n_4^{4,3}, n_{29}^{7,8}, n_{54}^{7,4}, n_1^{8,1})$
Todas-k-mensagens	$(n_7^{4,2}, scan_1, \dots, scan_{10}), (n_7^{4,2}, scan_{11}, \dots, scan_{20}), (n_1^{7,2}, msg_1, \dots, msg_{10}),$ $(n_1^{7,2}, msg_{11}, \dots, msg_{20}), (n_1^{7,3}, msg_1, \dots, msg_{10}), (n_1^{7,3}, msg_{11}, \dots, msg_{20}),$ $(n_3^{7,4}, laser_{scan_1}, \dots, laser_{scan_{10}}), (n_3^{7,4}, laser_{scan_{11}}, \dots, laser_{scan_{20}}),$ $(n_{21}^{7,8}, tf_1, \dots, tf_{10}), (n_{21}^{7,8}, tf_{11}, \dots, tf_{20}), (n_{29}^{7,1}, tf_1, \dots, tf_{10}),$ $(n_{29}^{7,1}, tf_{11}, \dots, tf_{20}), (n_1^{8,1}, tf_1, \dots, tf_{10}), (n_1^{8,1}, tf_{11}, \dots, tf_{20}),$ $(n_1^{9,1}, data_1, \dots, data_{10}), (n_1^{9,1}, data_{11}, \dots, data_{20}), (n_1^{10,1}, data_1, \dots, data_{10}),$ $(n_1^{10,1}, data_{11}, \dots, data_{20}), (n_{67}^{4,1}, tf_1, \dots, tf_{10}), (n_{67}^{4,1}, tf_{11}, \dots, tf_{20})$

Fonte: Elaborada pelo autor.

Os passos seguintes para a aplicação da abordagem são:

1. Gerar os casos de teste a partir dos cenários;
2. Avaliar as coberturas dos critérios para os casos de teste gerados no passo 1;
3. Inserir novos casos de teste para cobrir os elementos requeridos não cobertos.

Para a geração dos dados de teste considerou-se o conjunto de teste inicial (Tabela 10) baseado nos cenários (Quadro 4). A partir deste conjunto poderiam ser incluídos outros casos de teste capazes de aumentar a cobertura dos elementos requeridos, caso fosse necessário. Neste caso não foi necessário incluir novos casos de teste e assim o conjunto de teste final é o apresentado na Tabela 10. Após a execução dos casos de teste foram obtidas as coberturas ilustradas na Tabela 13.

Tabela 13 – Coberturas para os elementos requeridos para o Carina Parking

Critério	Qt elementos	Cobertura
Todos-nós-comunicação	25	100%
Todos-nós-publishers	15	100%
Todos-nós-subscribers	10	100%
Todos-pares-publishers-subscribers	27	100%
Todos-k-pares-publishers-subscribers	39	100%
Todas-k-mensagens	12	-

Fonte: Elaborada pelo autor.

Elementos requeridos não cobertos ocorreram nos casos em que um processo deixou de publicar ou de subscrever dados de um determinado tópico, como foi o caso dos tópicos *scan* ($n_7^{4,2}$ e $n_3^{7,4}$), *initialpose* ($n_1^{11,1}$ e $n_1^{7,2}$) e *map* ($n_{27}^{4,8}$ e $n_1^{7,3}$). Os dois primeiros não foram utilizados neste sistema, apesar de estarem definidos no código e se registrarem no processo Mestre para troca de dados. Durante a execução, o único processo que publica no tópico *map* é o *map_server* e o que subscreve é o *move_base*. Os pontos de comunicação nos processos *slam_gmapping* e *amcl* que publicam e subscrevem dados desse tópico não são executados. Dessa forma, as coberturas para os critérios todos-nós-comunicação, todos-nós-publishers, todos-nós-subscribers, todos-pares-publishers-subscribers e todos-k-pares-publishers-subscribers é 100%.

6.2.2 Discussão sobre o estudo de caso exploratório I

Uma característica do Carina Parking é que ele usa pacotes da pilha de navegação do ROS, de modo que, quase todo o código-fonte (7 processos) está incluído do núcleo do ROS. Dessa forma, somente os executáveis encontram-se na máquina em que o sistema está sendo executado. Essa limitação dificultou o teste do Carina Parking, pois, como a abordagem de teste também é estrutural é necessário o acesso ao código-fonte, não somente aos executáveis, para a geração do PSCFG. A solução para esse problema foi acessar os códigos-fonte dos processos que continham somente os executáveis a partir do repositório oficial do ROS⁴, mesmo local que os desenvolvedores baixam os pacotes para reuso. Em razão dessa particularidade, ficou inviável obter o rastreamento do fluxo de execução por meio de marcações no código sem uma

⁴ <https://github.com/ros>

ferramenta de suporte ao teste, já que parte do código precisaria ser recompilado e, como essa parte do código reside no *middleware*, testar esses pacotes se tornou também o teste do código do ROS. Para obter então medidas de cobertura, a solução foi obter o rastro da execução por meio da análise dos *logs* do sistema com a ajuda de ferramentas integradas ao ROS, como *rostopic*, *roscout*, dentre outras. Os elementos requeridos para os critérios que tratam do envio e recebimento de mensagens foram avaliados dessa forma. Para o critério todas-k-mensagens não foi possível a análise de cobertura, dado que as informações necessárias não são fornecidas pelas ferramentas utilizadas para avaliação da cobertura dos demais critérios.

Outra dificuldade na condução dos testes foi com relação ao oráculo do teste. A identificação do ponto objetivo fornecido pelo usuário e a avaliação da saída obtida quando comparada com a esperada. Apesar das coordenadas do ponto objetivo não serem fornecidas na interface gráfica da *Rviz*, quando a rota é calculada essa informação fica disponível em uma das telas do sistema, dado utilizado como base para as comparações.

Outro problema é que na interface da *Rviz* não há a delimitação das vagas do estacionamento para que o usuário escolha a vaga objetivo corretamente. Em decorrência dessa limitação o testador precisa inserir várias vezes um ponto objetivo até se certificar de que o ponto objetivo ocupa os limites de apenas uma vaga. Porém, observou-se que quando o sistema é executado, a saída (posição do robô estacionado) não é exatamente a esperada, talvez por uma limitação do planejador ou das dimensões do próprio robô em relação à rota. Considerou-se nos testes a saída esperada satisfatória quando o robô ocupa somente a vaga do estacionamento marcada como ponto objetivo.

Observou-se também com os cenários de teste que o sistema aceita como entrada qualquer sentido para o vetor objetivo fornecido pelo usuário. Como o sistema é para estacionamento, para essa configuração de mapa, seria conveniente limitar o sentido do vetor objetivo apenas para os sentidos existentes nas vagas, esquerda para direita (*theta* 0) e direita para esquerda (*theta* 8) a fim de maximizar o espaço do estacionamento. Quatro casos de teste, 24, 25, 26 e 27, testam essa situação com vetores objetivos com sentidos inclinados, porém, em nenhum dos casos o robô estacionou obedecendo o sentido objetivo (Figura 15), tendo esses casos de teste revelado falhas.

Dos 27 casos de teste especificados para cobrir os cenários de teste, 18 revelaram falhas. Notou-se que o planejador utilizado neste sistema para calcular a rota não se comporta da melhor forma possível, dadas as características desse ambiente e robô. Ao traçar a rota o algoritmo não considera as dimensões do robô e proximidade de obstáculos. Durante a execução da trajetória, em alguns casos, o robô colide com um obstáculo ou se aproxima de tal forma que ele não consegue continuar o trajeto iniciado. Isso aconteceu para o caso de teste 3, situação ilustrada na Figura 16. Acredita-se que esse comportamento do planejador seja atribuído pela vertente do reuso, no qual o planejador foi desenvolvido para traçar rotas para robôs de forma geral e o reuso nesse contexto deveria considerar essa particularidade (dimensões do robô e proximidade de obstáculos) pela criticidade do sistema.

Figura 15 – Execução para o cenário 10, caso de teste 25

Fonte: Elaborada pelo autor.

Figura 16 – Execução para o cenário 3, caso de teste 3

Fonte: Elaborada pelo autor.

Uma situação observada durante os testes com relação aos casos de teste 2, 3 e 4, que testam cenários de navegação do robô com dados de laser desativados ou manipulados, foi que independente dos dados do laser o robô atingiu o objetivo. Isso acontece porque o sistema utiliza

dois planejadores: 1) um global, para calcular a rota que o robô deve percorrer do ponto inicial até o ponto objetivo utilizando para isso o mapa do ambiente previamente obtido com informações sobre os obstáculos e; 2) um local, que usa os dados do laser para verificar se há algum obstáculo móvel na trajetória. Então, sem o sensor, o veículo segue a trajetória normalmente utilizando a rota calculada, pois nesse teste não há situações de inserir obstáculos móveis na trajetória do robô.

Outra falha revelada durante a execução dos casos de teste 11, 12, 18, 19, 20, 21 e 25 é que o robô encontra dificuldades para fazer curvas à esquerda. Curvas à direita ele realiza facilmente, mas quando necessário à esquerda raramente ele obtém sucesso, como ilustrado na Figura 17, em que ele não continua a trajetória.

Figura 17 – Execução para o cenário 10, caso de teste 24

Fonte: Elaborada pelo autor.

Foram identificadas possibilidades para obtenção de um melhor desempenho do sistema. O processo **amcl** é usado para a geração do mapa e após a conclusão dessa tarefa o processo continua executando, mesmo sem enviar ou receber dados de outros processos. Um processo em execução consome recursos do sistema e, dessa forma, sua desativação após a geração do mapa poderia melhorar o desempenho total do sistema.

Como conclusão do primeiro estudo, a abordagem de teste tem sua aplicabilidade indicada por ter apresentado um custo viável, considerando sua aplicação por um testador. Além disso, ela foi capaz de revelar a presença de defeitos no sistema em teste.

6.3 Estudo de caso exploratório II: RobAFISApplication

No segundo estudo a abordagem de teste proposta nesta tese foi novamente explorada com o objetivo de verificar sua aplicabilidade. A abordagem tem sua aplicabilidade indicada caso os resultados apontem um custo de aplicação viável e contribuições quanto a capacidade de revelar defeitos no sistema utilizado no estudo.

O RobAFISApplication é um sistema robótico de transporte de estoque que utiliza dois robôs do tipo Pioneer 3-DX. Este sistema foi desenvolvido seguindo as instruções da competição europeia RobAFIS⁵. O sistema simula um chão de fábrica composto por dois robôs, que inicialmente encontram-se em “stand by” em suas áreas de descanso. Quando o usuário seleciona uma ação, um ou os dois robôs deve(m): deslocar-se até a área de produção (áreas em cinza) e pegar o produto selecionado pelo usuário, deixá-lo na área de armazenamento (áreas em vermelho) para o tipo escolhido, em seguida parar em uma das três áreas de descanso (áreas em preto) e retornar ao modo “stand by” até que o usuário selecione outra ação. Um sistema de controle gerencia os dois robôs e a execução das tarefas. O sistema foi desenvolvida usando ROS e o simulador GAZEBO⁶. A Figura 18 ilustra o sistema em ambiente de simulação com a visualização fornecida pela ferramenta Rviz.

Figura 18 – RobAFIS em ambiente de simulação

Fonte: Elaborada pelo autor.

A Figura 19 mostra uma parte do grafo de comunicação do RobAFIS. O grafo completo é maior e para facilitar a visualização optou-se por exibir apenas uma parte dele.

RobAFIS possui 26 processos:

- *Application*, que gerencia os dois robôs e comunica-se usando *actions*;

⁵ <http://www.robafis.fr/RobAFIS>

⁶ <http://wiki.ros.org/gazebo>

Figura 19 – Parte do grafo de comunicação para o RobAFIS

Fonte: Elaborada pelo autor.

- *MapInformation* guarda informações do mapa do ambiente, nesse sistema ele não interage com outros nós;
- *Operation* é responsável pelo controle das tarefas para os dois robôs e interage somente com *actions*;
- *Robot1/PathPlanning* e *Robot2/PathPlanning* são planejadores para as rotas, porém não interagem com outros nós;
- *Robot1/TransportAgent* e *Robot2/TransportAgent* gerenciam os dois robôs com relação ao transporte dos materiais e somente interage com *actions*;
- *Robot1/Pioneer3AT_laserscan_tf* e *Robot2/Pioneer3AT_laserscan_tf* publicam dados no tópico *tf*, porém nenhum outro nó assina *tf*;
- *Robot1/BaseDriver* e *Robot2/BaseDriver* controlam a base do robô;
- *Robot1/Gazebo_Bridge* e *Robot2/Gazebo_Bridge* agem como intermediários entre o gazebo e o ROS;
- *Robot1/Localization* e *Robot2/Localization* controlam a posição do robô no ambiente;
- *Robot1/Navigator* e *Robot2/Navigator* são responsáveis pela navegação do robô;
- *Robot1/Gazebo_laserscan* e *Robot2/Gazebo_laserscan* publicam transformações do centro do robô para o laser;
- *Robot1/Mapper* e *Robot2/Mapper* gerenciam o mapeamento do ambiente;
- *Robot1/Object_manipulation* e *Robot2/Object_manipulation* controlam ações das garras dos robôs;
- *Gazebo* recebe e envia comandos de simulação para o robô, como exemplo, comandos relacionados a garra;
- *Robot1/Control* e *Robot2/Control* interagem e gerenciam os outros nós para alcançar o objetivo final.

6.3.1 Execução do teste - RobAFISApplication

Assim que o sistema inicializa é exibido um menu com três opções para o usuário: 0) sair do sistema; 1) criar mapa do ambiente ou; 2) selecionar produto. Caso o usuário selecione 0 o sistema finaliza. Caso o usuário escolha a opção 2, é impressa uma mensagem sugerindo primeiro mapear o ambiente, o que é possível com a opção 1. Após selecionar 1 e mapear o ambiente, o menu é novamente exibido com as três opções. Nessa fase o usuário pode sair do sistema, entrando com 0 ou selecionar um produto com a opção 2. Caso ele opte por inserir

1, uma mensagem informando que o mapa já foi criado é exibida. Dessa forma, com base na especificação do sistema são admitidas as seguintes entradas:

- Dados do sensor laser
- Dados de odometria
- Opção de ação no primeiro menu
- Opção de produto no segundo menu

A partir da especificação foram definidos os seguintes cenários de teste (Quadro 5):

Quadro 5 – Cenários para o RobAFIS

Cenário	Descrição do cenário
I	Verificar se um dos robôs atinge o objetivo sem dados de odometria
II	Verificar se um dos robôs atinge o objetivo sem dados do sensor laser
III	Verificar se um robô atinge o objetivo
IV	Verificar se os dois robôs atingem objetivos distintos
V	Verificar se os dois robôs atingem os mesmos objetivos seguidamente

Fonte: Elaborada pelo autor.

Com base na especificação do sistema e entradas é aplicado o critério particionamento de equivalência, gerando as seguintes classes para os cenários (Tabela 14).

Como esse sistema possui mais de uma tela para entrada dos dados, nesta tabela foi inserido o símbolo :: para marcar os limites entre uma execução e outra. Como exemplo, para o cenário 1 *verificar se os robôs atingem o objetivo sem dados de odometria* os dados de odometria foram desativados. Na primeira tela será exibido um menu para o usuário entrar com opções para sair do sistema ($M = 0$), mapear o ambiente ($M = 1$) e escolher produto ($M = 2$). Para verificar se o robô será capaz de navegar, cumprindo os objetivos, é necessário que o usuário selecione inicialmente mapear o ambiente, ação com $M = 1$. Após o mapeamento realizado o usuário selecionará outra ação, novamente será exibido o menu com as opções para sair do sistema ($M = 0$), mapear o ambiente ($M = 1$) e escolher produto ($M = 2$). Agora, o usuário deverá selecionar $M = 2$ para escolher um produto. Nesse caso, será exibido outro menu para o usuário selecionar o produto que deseja que o robô faça o transporte, produto 1 ($P = 1$) ou produto 2 ($P = 2$). Para as entradas geradas com base nas classes de equivalência as seguintes saídas deveriam ser analisadas:

- Mensagem informando que o sistema está disponível e exibição do primeiro menu;
- Mensagem para mapear o ambiente antes de solicitar um produto e exibição do primeiro menu;

Tabela 14 – Classes de equivalência para o RobAFIS

Cenário	Entrada	Id/Classe válida	Id/Classe inválida
I	L	CV ₁ : dados válidos	CI ₁ : dados válidos
	O	dados inválidos	dados válidos
	M	$M = 1, 2$	$M = 1, 2, 2$
	P	$P = 1 :: P = 2$	$P = 1, 2 :: P = 2, 1$
II	L	CV ₂ : dados inválidos	CI ₂ : dados válidos
	O	dados válidos	dados válidos
	M	$M = 1, 2$	$M = 1, 2, 2$
	P	$P = 1 :: P = 2$	$P = 1, 2 :: P = 2, 1$
III	L	CV ₃ : dados válidos	CI ₃ : dados válidos
	O	dados válidos	dados válidos
	M	$M = 1, 2$	$M = 1, 2, 2$
	P	$P = 1 :: P = 2$	$P = 1, 2 :: P = 2, 1$
IV	L	CV ₄ : dados válidos	CI ₄ : dados válidos
	O	dados válidos	dados válidos
	M	$M = 1, 2, 2$	$M = 1, 2, 2$
	P	$P = 1, 2 :: P = 2, 1$	$P = 1, 1 :: P = 2, 2$
V	L	CV ₅ : dados válidos	CI ₅ : dados válidos
	O	dados válidos	dados válidos
	M	$M = 1, 2, 2$	$M = 1, 2, 2$
	P	$P = 1, 2 :: P = 1, 2 / P = 2, 1 :: P = 2, 1$	$P = 1, 2 :: P = 2, 1 / P = 2, 1 :: P = 1, 2$

Fonte: Elaborada pelo autor.

Nota: L - dados do sensor laser, O - dados de odometria, M - ação no menu principal, P - opção de produto, :: - delimitador de execuções

- Mensagem informando que o mapa foi criado e exibição do segundo menu;
- Mensagem perguntando qual produto o usuário deseja selecionar para o transporte;
- Mensagem para entrada inválida;
- Produto entregue com sucesso.

A Tabela 15 ilustra o conjunto de casos de teste gerado com base nas classes de equivalência e saídas para o sistema. Durante a execução do RobAFIS são inseridas sequências de entradas em que a segunda opção depende da seleção da opção anterior e, em razão disso, a execução dos testes foi demorada por exigir reiniciar o sistema várias vezes e mapear o ambiente, já que na versão utilizada somente dois produtos (um de cada área) são produzidos em cada execução. Cada sequência de ações do usuário é representada na Tabela 15 por um bloco de entradas. Como exemplo, o cenário I, que gerou a classe de equivalência CV₁, contém o dado de teste 1, cuja execução à medida que as entradas são inseridas gera uma sequência de 4 saídas para o sistema.

Este conjunto de teste foi suficiente para os cenários de teste descritos no Quadro 5. Após esta fase, a abordagem prevê a aplicação dos critérios de teste que exploram a comunicação entre

Tabela 15 – Conjunto de casos de teste para o RobAFIS

Código			Entradas	Saída
C	CE	CT		
I	CV ₁	1	L = válido O = inválido M = 1, 2 P = 1 :: P = 2	Sistema disponível e exibição do primeiro menu Mapa criado e exibição do segundo menu Sistema solicita produto para o transporte Produto entregue com sucesso
II	CV ₂	2	L = inválido O = válido M = 1, 2 P = 1 :: P = 2	Sistema disponível e exibição do primeiro menu Mapa criado e exibição do segundo menu Sistema solicita produto para o transporte Produto entregue com sucesso
III	CV ₃	3	L = válido O = válido M = 1, 2 P = 1 :: P = 2	Sistema disponível e exibição do primeiro menu Mapa criado e exibição do segundo menu Sistema solicita produto para o transporte Produto entregue com sucesso
IV	CV ₄ , CI ₁ , CI ₂ , CI ₃	4	L = válido O = válido M = 1, 2, 2 P = 1, 2	Sistema disponível e exibição do primeiro menu Mapa criado e exibição do segundo menu Sistema solicita produto para o transporte Produto entregue com sucesso
		5	L = válido O = válido M = 1, 2, 2 P = 2, 1	Sistema disponível e exibição do primeiro menu Mapa criado e exibição do segundo menu Sistema solicita produto para o transporte Produto entregue com sucesso
	CI ₄	6	L = válido O = válido M = 1, 2, 2 P = 1, 1 :: P = 2, 2	Sistema disponível e exibição do primeiro menu Mapa criado e exibição do segundo menu Sistema solicita produto para o transporte Produto entregue com sucesso
V	CV ₅ , CI ₁ , CI ₂ , CI ₃	7	L = válido O = válido M = 1, 2, 2 P = 1, 2 :: P = 1, 2	Sistema disponível e exibição do primeiro menu Mapa criado e exibição do segundo menu Sistema solicita produto para o transporte Produto entregue com sucesso
		8	L = válido O = válido M = 1, 2, 2 P = 2, 1 :: P = 2, 1	Sistema disponível e exibição do primeiro menu Mapa criado e exibição do segundo menu Sistema solicita produto para o transporte Produto entregue com sucesso
	CI ₅	9	L = válido O = válido M = 1, 2, 2 P = 1, 2 :: P = 2, 1	Sistema disponível e exibição do primeiro menu Mapa criado e exibição do segundo menu Sistema solicita produto para o transporte Produto entregue com sucesso
		10	L = válido O = válido M = 1, 2, 2 P = 2, 1 :: P = 1, 2	Sistema disponível e exibição do primeiro menu Mapa criado e exibição do segundo menu Sistema solicita produto para o transporte Produto entregue com sucesso

Fonte: Elaborada pelo autor.

Nota: C - cenário, CE - classe de equivalência, CT - caso de teste, L - dados do sensor laser, O - dados de odometria, M - ação para o primeiro menu, P - opção de produto, :: - delimitador de execuções

os processos. Para a aplicação dos critérios, inicialmente foi gerado o PSCFG (Figura 20). Para a construção do PSCFG foram desconsideradas características de comunicação por serviços e *actions*, que não são o foco de estudo desta abordagem de teste. Também foram desconsiderados

tópicos que não são utilizados (quando nenhum processo publica ou subscreve).

Figura 20 – PSCFG para o RobAFIS

Fonte: Elaborada pelo autor.

A Tabela 16 contém os conjuntos obtidos a partir das definições no grafo que são utilizados para a geração dos requisitos de teste. Com base no PSCFG foram gerados os elementos requeridos (Tabela 17).

Tabela 16 – Informações geradas pelo modelo de teste para o RobAFIS

$n_p = 17, p_1 = \{m^{1,1}, m^{1,2}, m^{1,3}\}, p_2 = \{m^{2,1}, m^{2,2}\}, p_3 = \{m^{3,1}\},$ $p_4 = \{m^{4,1}, m^{4,2}\}, p_5 = \{m^{5,1}\}, p_6 = \{m^{6,1}\},$ $p_7 = \{m^{7,1}, m^{7,2}, m^{7,3}\}, p_8 = \{m^{8,1}, m^{8,2}, m^{8,3}, m^{8,4}\}, p_9 = \{m^{9,1}\},$ $p_{11} = \{m^{1,1}, m^{1,2}, m^{1,3}\}, p_{12} = \{m^{2,1}, m^{2,2}\}, p_{13} = \{m^{3,1}\},$ $p_{14} = \{m^{4,1}, m^{4,2}\}, p_{15} = \{m^{5,1}\}, p_{16} = \{m^{6,1}\},$ $p_{17} = \{m^{7,1}, m^{7,2}, m^{7,3}\}, p_{19} = \{m^{9,1}\},$ $Prog = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}, p_{16}, p_{17}, p_{19}\},$
$N^{1,1} = \{n_1^{1,1}\}, N^{1,2} = \{n_1^{1,2}\}, N^{1,3} = \{n_1^{1,3}\}, N^{11,1} = \{n_1^{11,1}\}, N^{11,2} = \{n_1^{11,2}\}, N^{11,3} = \{n_1^{11,3}\},$ $N^{2,1} = \{n_1^{2,1} \dots n_5^{2,1}\}, N^{12,1} = \{n_1^{12,1} \dots n_5^{12,1}\}, N^{2,2} = \{n_1^{2,2}\}, N^{12,2} = \{n_1^{12,2}\},$ $N^{3,1} = \{n_1^{3,1}\}, N^{13,1} = \{n_1^{13,1}\}, N^{4,1} = \{n_1^{4,1}\}, N^{14,1} = \{n_1^{14,1}\}, N^{4,2} = \{n_1^{4,2}\}, N^{14,2} = \{n_1^{14,2}\},$ $N^{5,1} = \{n_1^{5,1} \dots n_{10}^{5,1}\}, N^{15,1} = \{n_1^{15,1} \dots n_{10}^{15,1}\}, N^{6,1} = \{n_1^{6,1}\}, N^{16,1} = \{n_1^{16,1}\}, N^{7,1} = \{n_1^{7,1}\}, N^{17,1} =$ $\{n_1^{17,1}\}, N^{7,2} = \{n_1^{7,2}\}, N^{17,2} = \{n_1^{17,2}\},$ $N^{7,3} = \{n_1^{7,3}\}, N^{17,3} = \{n_1^{17,3}\}, N^{8,1} = \{n_1^{8,1}\}, N^{8,2} = \{n_1^{8,2}\}, N^{8,3} = \{n_1^{8,3}\}, N^{8,4} = \{n_1^{8,4}\}, N^{9,1} =$ $\{n_1^{9,1}\}, N^{19,1} = \{n_1^{19,1}\},$ $N^1 = N^{1,1} \cup N^{1,2} \cup N^{1,3}, N^2 = N^{2,1} \cup N^{2,2}$ $N^3 = N^{3,1}, N^4 = N^{4,1}$ $N^5 = N^{5,1}, N^6 = N^{6,1}, N^7 = N^{7,1} \cup N^{7,2} \cup N^{7,3}$ $N^8 = N^{8,1} \cup N^{8,2} \cup N^{8,3} \cup N^{8,4}, N^9 = N^{9,1},$ $N^{11} = N^{11,1} \cup N^{11,2} \cup N^{11,3}, N^{12} = N^{12,1} \cup N^{12,2}$ $N^{13} = N^{13,1}, N^{14} = N^{14,1} \cup N^{14,2}$ $N^{15} = N^{15,1}, N^{16} = N^{16,1}, N^{17} = N^{17,1} \cup N^{17,2} \cup N^{17,3}$ $N^{19,1} = \{n_1^{19,1}\},$ $N = N^1 \cup N^2 \cup N^3 \cup N^4 \cup N^5 \cup N^6 \cup N^7 \cup N^8 \cup N^9 \cup N^{11} \cup N^{12} \cup N^{13} \cup N^{14} \cup N^{15} \cup N^{16} \cup N^{17} \cup N^{19}$
$V^2 = \{ (v1, 'msg_in'), (v2, 'msg_in') \},$ $V^3 = \{ (v3, 'odomData') \},$ $V^4 = \{ (v4, 'laserData'), (v5, 'localization') \},$ $V^5 = \{ (v6, 'msg_in') \}, V^6 = \{ (v7, 'laserData') \}, V^7 = \{ (v8, 'state') \}, V^8 = \{ (v9, 'laserData') \},$ $V^9 = \{ (v9, 'localization') \}, V^{12} = \{ (v1, 'msg_in'), (v2, 'msg_in') \}, V^{13} = \{ (v3, 'odomData') \},$ $V^{14} = \{ (v4, 'laserData'), (v5, 'localization') \},$ $V^{15} = \{ (v6, 'msg_in') \},$ $V^{16} = \{ (v7, 'laserData') \},$ $V^{17} = \{ (v8, 'state') \},$ $V^{19} = \{ (v9, 'localization') \},$ $V = V^2 \cup V^3 \cup V^4 \cup V^5 \cup V^6 \cup V^7 \cup V^8 \cup V^9 \cup V^{12} \cup V^{13} \cup V^{14} \cup V^{15} \cup V^{16} \cup V^{17} \cup V^{19}$

Fonte: Elaborada pelo autor.

A Tabela 18 ilustra as coberturas obtidas pelos casos de teste representados na Tabela 15 em relação aos elementos requeridos.

Tabela 18 – Coberturas para os elementos requeridos para o RobAFIS

Critério	Qt. elementos	Cobertura
Todos-nós-comunicação	36	100%
Todos-nós-publishers	18	100%
Todos-nós-subscribers	19	100%
Todos-pares-publishers-subscribers	26	100%
Todos-k-pares-publishers-subscribers	40	100%
Todas-k-mensagens	38	100%

Fonte: Elaborada pelo autor.

Para dois critérios ocorreram elementos não executáveis, todos-k-pares-publishers-subscribers e todas-k-mensagens. Para o critério todos-k-pares-publishers-subscribers os elementos requeridos estão relacionados com mensagens a partir de uma função destrutora que não foi executada $((n_1^{1,1}, n_1^{2,2}, n_5^{2,1}, n_1^{3,1})$ e $((n_1^{11,1}, n_1^{12,2}, n_5^{12,1}, n_1^{13,1}))$ e caminhos não executados na ordem especificada pelo elemento requerido $(n_1^{1,2}, n_1^{2,2}, n_5^{2,1}, n_1^{3,1})$, $(n_1^{1,3}, n_1^{2,2}, n_5^{2,1}, n_1^{3,1})$, $(n_1^{11,1}, n_1^{12,2}, n_5^{12,1}, n_1^{13,1})$, e $(n_1^{11,2}, n_1^{12,2}, n_5^{12,1}, n_1^{13,1})$. Para o primeiro caso, a função destrutora executará quando o nó for finalizado, assim o elemento é executável. Para o segundo caso, o caminho especificado pelo elemento requerido não será executado por causa da ordem das mensagens. Conclui-se assim, que o primeiro elemento requerido não coberto é executável e os demais são não executáveis e, dessa forma, as coberturas para os elementos requeridos pelo critério todos-k-pares-publishers-subscribers atingem 100%.

Para o critério todas-k-mensagens algumas mensagens não chegam aos processos assíncronos $(n_1^{14,1})$ em razão da inicialização do sistema e outras mensagens foram recebidas com ordens diferentes das publicadas $(n_1^{2,2}$ e $n_1^{12,2})$. A forma de inicialização é uma característica do sistema, assim, o elemento requerido relacionado é não executável. As mensagens recebidas com ordens diferentes das publicadas resultam da comunicação *publish/subscribe* em que um nó pode acessar dados de uma fila em que vários processos publicam. A ordem em que as mensagens são colocadas na fila é estabelecida pela implementação *publish/subscribe*, com base nisto, considera-se essa ordem não determinística para o usuário e dessa forma, os elementos requeridos são não executáveis e a cobertura para o critério todas-k-mensagens atinge 100%.

6.3.2 Discussão sobre o estudo de caso exploratório II

Para este sistema o comportamento não determinístico ficou mais evidente. Uma mesma entrada gera diferentes saídas e comportamentos do robô, a depender do que ocorre durante a simulação. Esta particularidade motivou a condução de várias execuções (10) para o mesmo caso

de teste, a fim de obter os variados comportamentos do sistema para a mesma entrada. Optou-se por 10 por não ser um número não tão alto nem tão baixo para conduzir as repetições. Durante as execuções dos casos de teste várias saídas não esperadas foram obtidas, revelando falhas, como:

- robô descarta o produto no local correto, porém, na sequência o robô colide e arrasta o produto até outro local;
- robô deixa cair o produto, porém, ao não perceber ele continua seguindo a rota para a área de armazenamento e depois para a área de descanso sem notar que ele não estava de posse do produto durante o percurso;
- robô colide com o produto e não consegue encontrá-lo novamente por estar fora do campo de percepção do mesmo.

Confrontando esses resultados com os do teste de cobertura, é necessário averiguar se as ordens das mensagens durante o processamento não influenciaram em alguns desses comportamentos do sistema. Os cenários de teste contribuem nesta abordagem para uma maior validação dos requisitos do sistema e adequação do conjunto de teste, cuja eficácia foi notada durante a execução dos casos de teste em que diferentes comportamentos foram obtidos para os cenários, sinalizando defeitos no sistema.

Como conclusão do segundo estudo, a abordagem de teste teve novamente sua aplicabilidade indicada por permitir ser aplicada por um testador. Além disso, ela foi capaz de auxiliar a revelar defeitos no sistema em teste.

6.4 Relação critério tipo de defeito

A abordagem de teste aqui apresentada explora defeitos típicos em sistemas robóticos descritos na Seção 5.3. A utilização de cenários e critérios de teste funcional auxiliam na identificação de falhas oriundas de uma incompleta especificação do sistema. Os critérios de teste estrutural definidos visam identificar falhas relacionadas ao código, tais como:

- Critério todos-nós-comunicação e subconjuntos
 - Mensagens não enviadas ou não recebidas em nós específicos;
- Critério todos-pares-publishers-subscribers and todos-k-pares-publishers-subscribers
 - Mensagens não enviadas ou não recebidas em caminhos específicos;
- Critério todas-k-mensagens
 - Mensagens perdidas ou sobrescritas quando são utilizadas frequências incorretas entre processos comunicantes;

- Mensagens perdidas quando o tempo de processamento dos *callbacks* não é o ideal;
- Mensagens duplicadas;
- Mensagens corrompidas;

Fazendo um paralelo com a taxonomia de falhas apresentada em [Carlson, Murphy e Nelson \(2004\)](#), Seção 5.3, as falhas que podem ser reveladas pela abordagem relacionam-se com os tipos: falhas físicas incluindo as subcategorias efetoras, sensor, sistema de controle, comunicação e energia e falhas humanas incluindo as de projeto e de interação. Quanto à reparabilidade e impacto, as falhas podem ser reparáveis de campo ou não, bem como terminais ou não.

Da taxonomia de ameaças apresentada por [Lussier et al. \(2004\)](#) falhas, erros e defeitos podem ser identificados pela abordagem. Somente os defeitos de interação maliciosos, como exemplo, um ataque intencional, não são explorados pela abordagem de teste proposta. Os erros relacionados à odometria descritos em [Borenstein e Feng \(1996\)](#) e [Ribeiro \(1999\)](#) não são enfatizados pela abordagem de teste por serem específicos de características de odometria.

6.5 Comparação com outras abordagens de teste para robôs móveis

Foi conduzida uma comparação teórica entre a abordagem de teste proposta nesta tese e outras abordagens que utilizam o *middleware* ROS. O resultado está apresentado no Quadro 6. Para essa comparação foram consideradas características das abordagens de teste tradicionais, como:

1. Fase de teste: unidade, integração ou sistema;
2. Teste *ad-hoc* ou sistemático com a aplicação de critérios de teste;
3. Tipo de critério: funcional, estrutural ou baseado em defeitos (mutação);
4. Necessidade do código-fonte;
5. Definição de ferramenta de apoio ao teste.

Além disso, considerou-se também características específicas de teste para aplicações robóticas, como:

1. Teste conduzido na plataforma objetivo ou em ambiente de simulação e;
2. Se considerou dados de entrada provenientes de uma execução anterior do sistema (*log*).

Quadro 6 – Comparação entre as abordagens de teste aplicadas ao ROS

Característica	Abordagens de teste						
	(LUCAS; ELBAUM; ROSENBLUM, 2012)	(WEISSHARDT <i>et al.</i> , 2014)	(BIHLMAIER; WÖRN, 2014)	(LIER; LÜTKEBOHLE; WACHSMUTH, 2014)	(ARAIZA-ILLAN <i>et al.</i> , 2015)	(NEVES, 2015)	Abordagem Tese
Teste de unidade	X	X		X		X	
Teste de integração							X
Teste de sistema		X	X		X		
Teste de <i>hardware</i>		X					
Teste <i>ad-hoc</i>	X	X	X	X			
Critério funcional							X
Critério de fluxo de controle						X	X
Critério de fluxo de dados							X
Teste baseado em mutação						X	
Exigência do código-fonte	X	X	X		X	X	X
Ferramenta		X		X	X	X	
Ambiente real							
Ambiente de simulação	X	X	X	X	X	X	X
Utiliza <i>log</i>		X		X		X	

Fonte: Elaborada pelo autor.

Das abordagens comparadas, todas são aplicadas em ambiente de simulação. Apenas uma delas valida a parte do *hardware* do robô, as demais utilizam o *hardware* simulado. Apenas uma abordagem utiliza critérios de teste estrutural, que é fruto de uma tese de doutorado defendida em 2015, do mesmo grupo de pesquisa da autora desta tese. Quase todas as abordagens utilizam o código-fonte. Poucos estudos definem uma ferramenta de apoio ao teste.

Nesse sentido, esta tese de doutorado, representada na última coluna da tabela, contribui neste cenário com a proposição de uma abordagem de teste de integração, composta por critérios de teste funcional e estrutural. Os critérios estruturais definidos nesta tese são baseados no fluxo de controle e de dados. A abordagem de teste faz uso do código-fonte do sistema, que é executado em ambiente de simulação.

6.6 Considerações finais

Este capítulo apresentou os estudos de casos exploratórios conduzidos para validar a abordagem de teste proposta nesta tese de doutorado. Sobre a aplicabilidade da abordagem conclui-se que é válida sua aplicação, pois ela mostrou-se possível de ser utilizada na prática, além disso, a abordagem foi eficaz em auxiliar a revelar defeitos relacionados à comunicação e a troca de mensagens. A abordagem é adequada para ser utilizada em aplicações do domínio de sistemas robóticos que utilizam o padrão de comunicação *publish/subscribe*, pois contribui para auxiliar o testador na sistematização da atividade de teste nesse domínio.

CONCLUSÃO

A robótica móvel vem ganhando cada vez mais espaço na indústria, seja para auxiliar na realização de atividades humanas ou mesmo substituir os humanos em tarefas mais arriscadas. Porém, junto ao uso crescente, aumenta também a preocupação com a confiabilidade e segurança desses sistemas.

Esta pesquisa apresentou uma abordagem de teste de integração para sistemas robóticos móveis que explora a comunicação entre processos de um sistema distribuído, defendendo a tese de que essa abordagem é aplicável e auxilia na sistematização da atividade de teste.

Como apresentado no capítulo 1, o objetivo geral desta tese foi desenvolver uma abordagem de teste de integração para sistemas robóticos móveis. Em razão do domínio ser uma área muito ampla, incluindo robôs móveis terrestres, aquáticos e aéreos, concentrou-se nesta pesquisa em explorar os sistemas robóticos móveis terrestres, desenvolvidos utilizando o padrão de comunicação *publish/subscribe* atualmente implementado no *middleware* ROS.

A pesquisa teve início com a condução de estudos sobre os sistemas robóticos móveis, considerando o *background* da aluna de doutorado, visando adequar o que era conhecido, sobre o teste de programas concorrentes para utilizar no teste de sistemas robóticos. Para isso, um tempo foi dedicado para analisar códigos de sistemas robóticos móveis para possibilitar esse mapeamento de conceitos e similaridades de uma área para outra. Durante esse processo, foi conduzida uma revisão de literatura, utilizando como meio um estudo de mapeamento sistemático com o objetivo de reunir estudos sobre o domínio da pesquisa.

Tanto o estudo dos sistemas robóticos móveis, quanto o mapeamento, serviram de base para a proposição da abordagem de teste constituída de teste funcional e estrutural. Do teste funcional foram herdados os critérios de teste, que são utilizados para a geração de dados de teste a partir dos cenários de teste. Do teste estrutural, a abordagem é composta pela definição de um modelo e critérios de teste específicos para sistemas robóticos móveis. Optou-se por explorar teste de integração, que difere do convencional teste de unidades, pois o interesse nesta tese

de doutorado foi estudar o aspecto de sincronização e comunicação entre os componentes do sistema, bem como defeitos resultantes desse contexto.

Para avaliar a abordagem proposta foram aplicados dois estudos de caso exploratórios, utilizando sistemas do domínio de robôs móveis terrestres. Ambos os estudos tiveram como objetivo verificar a aplicabilidade da abordagem proposta.

Com o estudo de caso I observou-se que a abordagem de teste proposta é promissora por ser aplicável na prática e pode contribuir com a identificação de defeitos no sistema em teste. O tempo investido na condução desse estudo foi muito grande devido à construção do grafo, que foi realizada de forma quase que inteiramente manual, excetuando-se o apoio do ROS. Apesar disso, foi possível aplicar todos os passos da abordagem, determinando-a como aplicável. Somando-se a este fato a abordagem revelou alguns defeitos no sistema. Outro estudo de caso foi conduzido para confirmação da viabilidade da abordagem de teste na prática. Com a condução do estudo de caso exploratório II reafirmou-se a contribuição da abordagem quanto a aplicabilidade, além de ter sido possível revelar defeitos também no segundo sistema testado.

Dessa forma, com a condução dos dois estudos exploratórios, considera-se ter obtido evidências de que a abordagem de teste proposta é aplicável na prática e auxilia na sistematização da atividade de teste de sistemas robóticos móveis. Além disso, ela pode contribuir com a redução do número de defeitos no produto final.

7.1 Contribuições e limitações da pesquisa

Esta tese apresenta duas principais contribuições para o teste de *software* no domínio de robótica móvel: um estudo sobre o mapeamento de conceitos do teste de programas concorrentes para o teste de robôs móveis e a definição de uma abordagem de teste, composta por uma parte funcional e outra estrutural.

A primeira contribuição se baseou no *background* da aluna de doutorado, visando adequar o que era conhecido por ela sobre o teste de programas concorrentes para adequar e utilizar no teste de sistemas robóticos móveis. Esse estudo serviu de base para a proposição do modelo de teste, grafo e critérios de teste estruturais da segunda contribuição.

A segunda contribuição foi a proposição de uma abordagem de teste, composta por teste funcional, com a especificação dos cenários de teste, e teste estrutural com um modelo e critérios de teste. Os cenários de teste visam validar as funcionalidades do sistema em teste, bem como a geração de dados de teste. A ideia de propor a definição de cenários de teste foi facilitar a geração de dados de teste nesse domínio, contemplando o que se espera na especificação do sistema em teste. O modelo de teste visa validar no código a comunicação entre os processos do sistema, servindo de base para a geração dos elementos requeridos, gerados a partir dos critérios estruturais. Estudos de caso exploratórios, utilizando sistemas robóticos foram conduzidos para

avaliar a abordagem de teste proposta.

Outras contribuições do trabalho foram: 1) exploração do ROS, *middleware* amplamente usado pela comunidade de desenvolvedores de sistemas robóticos, que carece de estudos mais aprofundados e; 2) utilização de dois diferentes simuladores para sistemas robóticos, durante a condução dos testes.

Algumas limitações deste trabalho foram identificadas e são listadas a seguir:

1. Ambos os estudos de caso exploratórios foram conduzidos utilizando sistemas desenvolvidos em ambiente acadêmico e isso implica em uma ameaça à validade dos resultados, do ponto de vista da generalização. Apesar disso, um aspecto importante é que os sistemas não foram desenvolvidos pela aluna, o que é importante para evitar um viés durante a atividade de teste;
2. Os estudos de caso exploraram apenas sistemas robóticos móveis terrestres, por serem eles os principais motivadores do trabalho e os que a aluna teve acesso;
3. Foram utilizados somente dois sistemas para validação da abordagem de teste. Um número maior de sistemas com tamanhos e características diferenciadas possibilitaria uma generalização dos resultados do estudo. Para isso, uma ferramenta que automatize a abordagem se faz necessária;
4. O modelo de teste proposto considera apenas 1 tipo de comunicação utilizado nos sistemas robóticos desenvolvidos utilizando o ROS, o modelo de comunicação *publish/subscribe*. Essa limitação não permitiu testar a comunicação entre os processos para outros tipos de comunicação como *actions* e serviços. Estender a abordagem para abarcar os demais tipos de comunicação poderia revelar outros tipos de defeitos.

7.2 Lições aprendidas

Durante a condução desta pesquisa algumas atividades não foram executadas como inicialmente planejadas e alguns desafios não previstos precisaram ser superados.

No início da pesquisa, um dos maiores desafios do projeto, foi com relação ao acesso aos sistemas robóticos. Os sistemas alvo da abordagem proposta são normalmente produtos comerciais, cujo código não é *open source* ou disponibilizado em repositórios. Para implementar um sistema, utilizando pacotes de acesso público disponíveis nos repositórios online, é necessário um conhecimento na área de desenvolvimento de aplicações robóticas, ambientes de desenvolvimento, simulação, etc., perfil que não era da aluna que estava desenvolvendo a tese. Somente com a disponibilização de um sistema a partir do Laboratório de Robótica Móvel é que a aluna teve acesso ao sistema e conseguiu iniciar o trabalho de ordem prática.

7.3 Oportunidades para trabalhos futuros

Há diversas pesquisas que podem surgir a partir dos estudos conduzidos durante este doutorado e apresentados nesta tese. Um estudo mais imediato seria a automatização da ferramenta de teste para apoiar a aplicação da abordagem de teste em outros estudos de caso. Esta tarefa é fundamental para que a abordagem de teste seja efetivamente utilizada pela comunidade de robótica que desenvolve os programas alvo.

Outra atividade interessante é a condução de estudos de caso com outros tipos de robôs, como aquáticos e aéreos, que também pertencem ao domínio de robótica móvel, mas que não foram estudados durante este doutorado em razão da oportunidade.

Uma variedade de estudos poderiam ser conduzidos com diversos objetivos, como: avaliação da efetividade da abordagem em revelar defeitos, que poderia ser medida pelo número de defeitos semeados e revelados; custo em revelar defeitos, medido pelo número de dados de teste utilizados; desempenho do sistema, medido pelo número de componentes que o *software* contém e os efetivamente usados para realizar determinadas tarefas, dentre outros. Uma possibilidade para a semeadura de defeitos seria, por exemplo, utilizar operadores de mutação.

Outro trabalho futuro poderia ser implementar uma abordagem para geração de dados de teste, em conformidade com a ferramenta que apoie a execução do teste.

Integrar esta abordagem com o trabalho proposto em [Neves \(2015\)](#). Neste caso, a ferramenta existente, a PATECa, funciona com base em dados de teste provenientes de uma execução anterior, abordagem um pouco diferente da proposta nesta tese, cujos dados de teste são provenientes do simulador. Porém, como a PATECa já possui a parte de geração de dados com base em *logs*, seriam adicionadas a parte do PSCFG, gerado com base nos GFCs, que a ferramenta já gera e a implementação dos critérios de teste aqui definidos.

REFERÊNCIAS

AFZAL, W.; TORKAR, R.; FELDT, R. A systematic review of search-based testing for non-functional system properties. **Information and Software Technology**, Butterworth-Heinemann, Newton, MA, USA, v. 51, n. 6, p. 957–976, jun. 2009. Citado na página 73.

ALLEN, F. E. Control flow analysis. In: **Proceedings of a Symposium on Compiler Optimization**. New York, NY, USA: ACM, 1970. p. 1–19. Citado na página 84.

ALMASI, G. S.; GOTTLIEB, A. **Highly Parallel Computing (2Nd Ed.)**. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994. ISBN 0-8053-0443-6. Citado nas páginas 44 e 79.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. [S.l.]: Cambridge University Press, 2008. Citado na página 40.

ANDREWS, A.; ABDELGAWAD, M.; GARIO, A. Active world model for testing autonomous systems using CEFMS. In: **12th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA)**. Ottawa, ON, Canada: [s.n.], 2015. v. 1514, p. 1 – 10. Citado nas páginas 58, 62, 63 e 64.

ANDREWS, G. R.; SCHNEIDER, F. B. Concepts and notations for concurrent programming. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 15, n. 1, p. 3–43, 1983. ISSN 0360-0300. Citado na página 45.

ARAIZA-ILLAN, D.; WESTERN, D.; PIPE, A.; EDER, K. Coverage-driven verification: An approach to verify code for robots that directly interact with humans. In: PITERMAN, N. (Ed.). **Hardware and Software: Verification and Testing: 11th International Haifa Verification Conference, HVC, Haifa, Israel, Proceedings**. Cham: Springer International Publishing, 2015. Citado nas páginas 58, 60, 62, 63, 65, 76, 80, 81 e 124.

ARAUJO, A. G. **ROSint - integration of a mobile robot in ROS architecture**. 82 p. Dissertação (Mestrado) — University of Coimbra, Coimbra, Portugal, 2012. Citado na página 30.

ARNOLD, J.; ALEXANDER, R. Testing autonomous robot control software using procedural content generation. In: **International Conference on Computer Safety, Reliability, and Security**. New York, NY, USA: Springer-Verlag New York, Inc., 2013. (SAFECOMP 2013), p. 33–44. ISBN 978-3-642-40792-5. Citado nas páginas 58, 62, 63 e 74.

ARNOLD, T. R.; FUSON, W. A. Testing in a perfect world. **Commun. ACM**, New York, NY, USA, v. 37, n. 9, p. 78–86, set. 1994. Citado na página 42.

BECHINI, A.; CUTAJAR, J.; PRETE, C. A tool for testing of parallel and distributed programs in message-passing environments. In: **Electrotechnical Conference, 1998. MELECON 98., 9th Mediterranean**. [S.l.: s.n.], 1998. v. 2, p. 1308 –1312 vol.2. Citado na página 45.

BELLI, F.; HOLLMANN, A.; PADBERG, S. Communication sequence graphs for mutation-oriented integration testing. In: **International Conference on Secure Software Integration and Reliability Improvement**. Washington, DC, USA: [s.n.], 2009. p. 387–392. Citado nas páginas 58, 59, 60, 61, 62, 63, 67 e 76.

BEN-ASHER, Y.; EYTANI, Y.; FARCHI, E.; UR, S. Noise makers need to know where to be silent - producing schedules that find bugs. In: **ISOLA '06: Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation**. Washington, DC, USA: IEEE Computer Society, 2006. p. 458–465. Citado na página 32.

BENGTSSON, J.; YI, W. Timed automata: Semantics, algorithms and tools. In: _____. **Lectures on Concurrency and Petri Nets: Advances in Petri Nets**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 87–124. Citado na página 71.

BEYDEDA, S.; GRUHN, V. An integrated testing technique for component-based software. In: **International Conference on Computer Systems and Applications**. Washington, DC, USA: [s.n.], 2001. p. 328–. Citado na página 43.

BIGGS, G. Applying regression testing to software for robot hardware interaction. In: **International Conference on Robotics and Automation**. Anchorage, AK, United states: [s.n.], 2010. p. 4621 – 4626. Citado nas páginas 58, 62, 63 e 67.

BIHLMAIER, A.; WÖRN, H. Robot unit testing. In: **International Conference on Simulation, Modeling, and Programming for Autonomous Robots**. New York, NY, USA: Springer-Verlag New York, Inc., 2014. (SIMPAN 2014), p. 255–266. Citado nas páginas 58, 62, 63, 74, 76, 80 e 124.

BINDER, R. V. **Testing Object-oriented Systems: Models, Patterns, and Tools**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. Citado na página 42.

BORENSTEIN, J.; FENG, L. Measurement and correction of systematic odometry errors in mobile robots. **IEEE Transactions on Robotics and Automation**, v. 12, n. 6, p. 869–880, 1996. Citado nas páginas 82 e 123.

BRITO, M.; SANTOS, M.; SOUZA, P.; SOUZA, S. Integration testing criteria for mobile robotic systems. In: **International Conference on Software Engineering and Knowledge Engineering, SEKE**. [S.l.: s.n.], 2015. v. 2015-January, p. 182–187. Citado nas páginas 58, 62, 63, 70, 76 e 92.

BRITO, M.; SANTOS, M.; SOUZA, S.; SOUZA, P. A systematic mapping of data generation for integration software testing. In: **13th Empirical Software Engineering Latin American Workshop (ESELAW 2016)**. [S.l.: s.n.], 2016. p. 1–14. Citado na página 76.

BRITO, M. A. S.; SOUZA, S. R. S.; SOUZA, P. S. L. An empirical evaluation of the cost and effectiveness of structural testing criteria for concurrent programs. In: **International Conference on Computational Science**. Barcelona, Spain: [s.n.], 2013. Citado nas páginas 45 e 46.

BRITO, P.; LEMOS, R. D.; RUBIRA, C.; MARTINS, E. Architecting fault tolerance with exception handling: Verification and validation. **Journal of Computer Science and Technology**, v. 24, n. 2, p. 212–237, 2009. Citado nas páginas 58, 62, 63, 70, 76 e 84.

- BROOKS, A.; KAUPP, T.; MAKARENKO, A.; WILLIAMS, S.; OREBÄCK, A. Orca: A component model and repository. In: BRUGALI, D. (Ed.). **Software Engineering for Experimental Robotics**. [S.l.]: Springer Berlin Heidelberg, 2007. v. 30, p. 231–251. Citado na página 50.
- BRUYNINCKX, H. Open robot control software: the OROCOS project. In: **International Conference on Robotics and Automation**. [S.l.: s.n.], 2001. v. 3, p. 2523–2528. Citado na página 50.
- BUI, T.-Q.; AKTOUF, O.-E.-K. On-line testing of software components for diagnosis of embedded systems. In: **World Academy of Science Engineering and Technology**. [S.l.: s.n.], 2007. v. 22, p. 330–336. Citado nas páginas 58, 62, 63 e 66.
- CARLSON, J.; MURPHY, R. R.; NELSON, A. Follow-up analysis of mobile robot failures. In: **International Conference on Robotics and Automation, ICRA**. [S.l.: s.n.], 2004. v. 5, p. 4987–4994 Vol.5. Citado nas páginas 81, 82, 83, 96 e 123.
- CARVALHO, A.; KOWALTOWSKI, T. **Atualizações em informática**. [S.l.]: PUC-Rio, 2009. Citado na página 48.
- CARVER, R.; LEI, Y. A general model for reachability testing of concurrent programs. In: **International Conference on Formal Engineering Methods**. [S.l.: s.n.], 2004. v. 3308, p. 76–98. Citado na página 45.
- CHEN, H. Y.; TSE, T. H.; CHAN, F. T.; CHEN, T. Y. In black and white: an integrated approach to class-level testing of object-oriented programs. **ACM Trans. Softw. Eng. Methodol.**, New York, NY, USA, v. 7, n. 3, p. 250–295, jul. 1998. ISSN 1049-331X. Citado na página 42.
- CHEN, Q.; WANG, L.; YANG, Z.; STOLLER, S. HAVE: Detecting atomicity violations via integrated dynamic and static analysis. In: CHECHIK, M.; WIRSING, M. (Ed.). **Fundamental Approaches to Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2009. v. 5503, p. 425–439. Citado na página 30.
- CHEN, Z.; LI, X.; CHEN, J. Y.; ZHONG, H.; QIN, F. SyncChecker: Detecting synchronization errors between MPI applications and libraries. In: **Parallel Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2012. p. 342–353. Citado na página 32.
- CHUNG, Y. K.; HWANG, S.-M. Software testing for intelligent robots. In: **International Conference on Control, Automation and Systems, ICCAS**. [S.l.: s.n.], 2007. p. 2344 – 2349. Citado nas páginas 58, 63, 66 e 76.
- COLANZI, T. E. **Uma abordagem integrada de desenvolvimento e teste de software baseada na UML**. Dissertação (Mestrado) — ICMC/USP, São Carlos, SP, 1999. Citado na página 42.
- COOMBES, M.; EATON, W.; MCAREE, O.; CHEN, W.-H. Development of a generic network enabled autonomous vehicle system. In: **International Conference on Control, CONTROL**. [S.l.: s.n.], 2014. p. 621 – 627. Citado nas páginas 58, 62, 63 e 72.
- COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T.; BLAIR, G. **Distributed systems concepts and design**. New York, USA: Addison-Wesley, 2001. Citado nas páginas 80 e 83.
- COWARD, P. D. A review of software testing. **Information and Software Technology**, Butterworth-Heinemann, Newton, MA, USA, v. 30, n. 3, p. 189–198, 1988. ISSN 0950-5849. Citado na página 29.

DAMM, W.; HUNGAR, H.; JOSKO, B.; PEIKENKAMP, T.; STIERAND, I. Using contract-based component specifications for virtual integration testing and architecture design. In: **Design, Automation Test in Europe Conference Exhibition**. [S.l.: s.n.], 2011. p. 1023–1028. Citado nas páginas 58, 62, 63, 65 e 84.

DELAMARO, M. E. **Mutação de Interface: Um Critério de Adequação Inter-Procedural para o Teste de Integração**. Tese (Doutorado) — Instituto de Física de São Carlos da Universidade de São Paulo (ICMC/USP), 1997. Citado na página 42.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao Teste de Software**. [S.l.: Elsevier, 2007. (Campus, v. 394). Citado nas páginas 35, 36, 43, 65, 74, 82 e 83.

DEMILLO, R. A. **Software testing and evaluation**. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1986. ISBN 0-805-32535-2. Citado na página 38.

DOLIN, B.; BENNETT, F.; RIEFFEL, E. Methods for evolving robust distributed robot control software: Coevolutionary and single population techniques. In: **Proceedings - NASA/DoD Conference on Evolvable Hardware, EH**. [S.l.: s.n.], 2001. v. 2001-January, p. 21 – 29. Citado nas páginas 58, 62, 63 e 73.

DUDEK, G.; JENKIN, M. **Computational Principles of Mobile Robotics**. 2nd. ed. [S.l.: Cambridge University Press, 2010. Citado na página 48.

DUVALL, P.; MATYAS, S. M.; GLOVER, A. **Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)**. [S.l.: Addison-Wesley Professional, 2007. ISBN 0321336380. Citado na página 69.

EBERT, C.; SALECKER, J. a. Guest editors' introduction: Embedded software technologies and trends. **IEEE Software**, v. 26, n. 3, p. 14 –18, 2009. Citado na página 48.

EDWARDS, S. H. A framework for practical, automated black-box testing of component-based software. **Software Testing, Verification and Reliability**, John Wiley & Sons, Ltd., v. 11, n. 2, p. 97–111, 2001. Citado na página 43.

_____. Toward reflective metadata wrappers for formally specified software components. In: **Workshop on specification and verification of component-based systems**. [S.l.: s.n.], 2001. p. 14–21. Citado na página 43.

ENDO, A. T.; SIMÃO, A. S.; SOUZA, S. R. S.; SOUZA, P. S. L. Web services composition testing: A strategy based on structural testing of parallel programs. In: **TaicPart: Testing Academic & Industrial Conference - Practice and Research Techniques**. Windsor: [s.n.], 2008. p. 3–12. Citado nas páginas 45 e 46.

ERNITS, J.; HALLING, E.; KANTER, G.; VAIN, J. Model-based integration testing of ROS packages: A mobile robot case study. In: **European Conference on Mobile Robots, ECMR**. [S.l.: s.n.], 2015. Citado nas páginas 58, 60, 62, 63, 71 e 76.

EUGSTER, P. T.; FELBER, P. A.; GUERRAOUI, R.; KERMARREC, A.-M. The many faces of publish/subscribe. **ACM Computing Surveys**, New York, NY, USA, v. 35, n. 2, p. 114–131, jun. 2003. Citado nas páginas 30 e 84.

FAN, X.; SHU, J.; LIU, L.; LIANG, Q. Test case generation from UML subactivity and activity diagram. In: **Int. Symposium on Electronic Commerce and Security**. [S.l.: s.n.], 2009. v. 2, p. 244–248. Citado nas páginas 30, 58, 62 e 75.

FERNANDES, L. C.; SOUZA, J. R.; PESSIN, G.; SHINZATO, P. Y.; SALES, D.; MENDES, C.; PRADO, M.; KLASER, R.; MAGALHÃES, A. C.; HATA, A.; PIGATTO, D.; BRANCO, K. C.; JR., V. G.; OSORIO, F. S.; WOLF, D. F. CaRINA intelligent robotic car: Architectural design and applications. **Journal of Systems Architecture**, v. 60, n. 4, p. 372 – 392, 2014. Citado nas páginas 30 e 50.

FIN, A.; FUMMI, F.; SIGNORETTO, D. The use of SystemC for design verification and integration test of IP-cores. In: **14th Annual IEEE International ASIC/SOC Conference**. [S.l.: s.n.], 2001. p. 76–80. Citado nas páginas 58, 62, 63 e 72.

FRIEDMANN, M.; PETERSEN, K.; STRYK, O. von. Scalable and adequate simulation for motion and sensors of heterogeneous teams of autonomous mobile robots. In: **Intl. Conf. on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)**. Venice, Italy: [s.n.], 2008. Citado na página 67.

GERKEY, B.; VAUGHAN, R.; STOY, K.; HOWARD, A.; SUKHATME, G.; MATARIC, M. Most valuable player: a robot device server for distributed control. In: **International Conference on Intelligent Robots and Systems**. [S.l.: s.n.], 2001. v. 3, p. 1226–1231 vol.3. Citado na página 50.

GLIGORIC, M.; JAGANNATH, V.; MARINOV, D. MuTMuT: Efficient exploration for mutation testing of multithreaded code. In: **International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2010. p. 55 –64. Citado na página 45.

GUNDERSON, J. P.; GUNDERSON, L. F. "what do you do with a drunken robot?": In situ performance measurements of intelligent mobile robots. In: **10th Performance Metrics for Intelligent Systems Workshop**. New York, NY, USA: [s.n.], 2010. p. 154–160. Citado nas páginas 58, 62, 63 e 71.

HARROLD, M. J.; SOFFA, M. L. Selecting and using data for integration testing. **IEEE Software**, v. 8, n. 2, p. 58–65, mar. 1991. Citado na página 41.

HARTMANN, J.; IMOBERDORF, C.; MEISINGER, M. UML-based integration testing. In: **International symposium on software testing and analysis**. New York, NY, USA: [s.n.], 2000. p. 60–70. Citado na página 44.

HAUSEN, A. C.; VERGILIO, S. R.; SOUZA, S.; SOUZA, P.; aO, A. S. A tool for structural testing of MPI programs. In: **Latin-American Test Workshop - LATW**. Cuzco: [s.n.], 2007. v. 1. Citado nas páginas 45 e 46.

HERZOG, O. Static analysis of concurrent processes for dynamic properties using petri nets. **Instrument Maintenance Management**, v. 70, p. 66–90, 1979. Citado na página 45.

HONKA, H.; KATTILAKOSKI, M. A simulation-based system for testing real-time embedded software in the host environment. **Microprocessing and Microprogramming**, v. 32, n. 1-5, p. 127–134, 1991. Citado nas páginas 58, 62, 63 e 66.

HOSSAIN, M. I.; LEE, W. J. A scalable integration testing approach considering independent usage pattern of global variables. **Int. Journal of Software Eng. and its Applications**, v. 7, n. 3, p. 195–204, 2013. Citado nas páginas 58, 62, 63 e 69.

HOWDEN, W. E. Methodology for the generation of program test data. **IEEE Trans. Comput.**, IEEE Computer Society, Washington, DC, USA, v. 24, n. 5, p. 554–560, 1975. ISSN 0018-9340. Citado na página 29.

- HURA, D.; DIMMICH, M. A method facilitating integration testing of embedded software. In: **9th International Workshop on Dynamic Analysis**. [S.l.: s.n.], 2011. p. 7–11. Citado nas páginas 58, 62, 63 e 68.
- HWANG, G.-H.; TAI, K. chung; HUANG, T. lu. Reachability testing: An approach to testing concurrent software. **International Journal of Software Engineering and Knowledge Engineering**, v. 5, p. 493–510, 1995. Citado na página 63.
- JAGANNATH, V.; GLIGORIC, M.; LAUTERBURG, S.; MARINOV, D.; AGHA, G. Mutation operators for actor systems. In: **International Conference on Software Testing, Verification, and Validation Workshops**. [S.l.: s.n.], 2010. p. 157–162. Citado na página 45.
- JANSCHKE, K.; MOROZOV, A. Dependability aspects of model-based systems design for mechatronic systems. In: **International Conference on Mechatronics, ICM**. [S.l.: s.n.], 2015. p. 15–22. Citado nas páginas 58, 62, 63 e 70.
- JORGENSEN, P. C.; ERICKSON, C. Object-oriented integration testing. **Commun. ACM**, New York, NY, USA, v. 37, n. 9, p. 30–38, set. 1994. Citado na página 42.
- JOUNG, T.-H.; CHOI, H.-S.; JUNG, S.-K.; SAMMUT, K.; HE, F. Verification of CFD analysis methods for predicting the drag force and thrust power of an underwater disk robot. **International Journal of Naval Architecture and Ocean Engineering**, v. 6, n. 2, p. 269–281, 2014. Citado nas páginas 58, 62, 63 e 71.
- JOYEUX, S.; ALBIEZ, J. Robot development: from components to systems. In: **6th National Conference on Control Architectures of Robots**. Grenoble, France: [s.n.], 2011. p. 1–15. Citado nas páginas 50 e 51.
- JUN, L.; MOORE, K. A runtime and analysis framework support for unit component testing in distributed systems. In: **Annual Hawaii International Conference on System Sciences**. [S.l.: s.n.], 2007. Citado nas páginas 58, 62, 63, 66 e 76.
- JUNG, M. Y.; DEGUET, A.; KAZANZIDES, P. A component-based architecture for flexible integration of robotic systems. In: **International Conference on Intelligent Robots and Systems**. [S.l.: s.n.], 2010. p. 6107–6112. Citado na página 50.
- KANER, C. **An introduction to scenario testing**. [S.l.], 2003. University Florida Tech. Citado nas páginas 70, 85 e 88.
- KANG, J.-S.; CHOI, H.-S.; MAENG, S.-W.; KIM, S.-W.; PARK, H.-S. SITAT: Simulation-based interface testing automation tool for robot software component. **Journal of Institute of Control, Robotics and Systems**, v. 16, n. 6, p. 608 – 616, 2010. Citado nas páginas 30, 58, 62, 63 e 67.
- KANG, J. S.; PARK, H. S. RPIST: Required and provided interface specification-based test case generation and execution methodology for robot software component. In: **Int. Conference on Ubiquitous Robots and Ambient Intelligence**. [S.l.: s.n.], 2011. p. 647–651. Citado nas páginas 30, 58, 62, 63 e 68.
- _____. Web-based automated black-box testing framework for component based robot software. In: **Proceedings of the ACM Conference on Ubiquitous Computing**. [S.l.: s.n.], 2012. Citado nas páginas 58, 62, 63 e 68.
- KAWAGUCHI, A. B. K.; CROY, R. **Jenkins – an extensible open source continuous integration server**. 2017. Disponível em: <<http://jenkins-ci.org/>>. Citado na página 80.

KIM, W.; SON, H.; KIM, R. A study on test case generation based on state diagram in modeling and simulation environment. **Communications in Computer and Information Science**, p. 298–305, 2011. Citado nas páginas 58, 62, 63 e 74.

KITCHENHAM, B.; CHARTERS, S. **Guidelines for performing Systematic Literature Reviews in Software Engineering**. [S.l.], 2007. School of Comp. Science and Mathematics. Citado na página 55.

KOPPOL, P. V.; CARVER, R. H.; TAI, K.-C. Incremental integration testing of concurrent programs. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 28, n. 6, p. 607–623, 2002. ISSN 0098-5589. Citado na página 45.

KOPPOL, P. V.; TAI, K.-C. An incremental approach to structural testing of concurrent software. In: **International symposium on software testing and analysis**. New York, NY, USA: [s.n.], 1996. p. 14–23. Citado na página 45.

KOSTAL, I. A.NET application searching for data in a log file of the KUKA industrial welding robot. In: **16th International Conference on Mechatronics, Mechatronika**. [S.l.: s.n.], 2014. p. 656 – 661. Citado nas páginas 58, 62, 63 e 75.

KRAWCZYK, H.; WISZNIEWSKI, B. **A Method for Determining Testing Scenarios for Parallel and Distributed Software**. [S.l.], 1996. SEPP Progress Report n. 5, Technical University of Gdansk. Citado na página 45.

LABICHE, Y.; THÉVENOD-FOSSE, P.; WAESELYNCK, H.; DURAND, M.-H. Testing levels for object-oriented software. In: **22nd international conference on Software engineering**. New York, NY, USA: [s.n.], 2000. p. 136–145. Citado na página 44.

LASKI, J. W.; KOREL, B. A data flow oriented program testing strategy. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 9, n. 3, p. 347–354, 1983. ISSN 0098-5589. Citado na página 29.

LEE, K.; KANG, J.; PARK, H. DocT - document management and testing tool for robot software. In: **11th International Conference on Ubiquitous Robots and Ambient Intelligence, URAI**. [S.l.: s.n.], 2014. p. 413–416. Citado nas páginas 58, 62, 63 e 69.

LEE, N.; KIM, T.; CHA, S. Construction of global finite state machine for testing task interactions written in message sequence charts. In: **International conference on Software engineering and knowledge engineering**. [S.l.: s.n.], 2002. v. 27, p. 369–376. Citado nas páginas 58, 62, 63, 70, 76 e 84.

LEI, Y.; CARVER, R. H. Reachability testing of concurrent programs. **IEEE Transactions on Software Engineering**, v. 32, n. 6, p. 382–403, 2006. Citado nas páginas 45 e 63.

LEI, Y.; CARVER, R. H.; KACKER, R.; KUNG, D. A combinatorial testing strategy for concurrent programs. **Software Testing Verification and Reliability**, v. 17, n. 4, p. 207–225, 2007. Citado na página 45.

LEMOS, O. A. L.; FRANCHIN, I. G.; MASIERO, P. C. Integration testing of object-oriented and aspect-oriented programs: A structural pairwise approach for Java. **Sci. Comput. Program.**, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 74, n. 10, p. 861–878, ago. 2009. ISSN 0167-6423. Citado na página 42.

LEUNG, H. K. N.; WHITE, L. A study of integration testing and software regression at the integration level. In: **Conference on Software Maintenance**. [S.l.: s.n.], 1990. p. 290–301. Citado na página 39.

LIER, F.; LÜTKEBOHLE, I.; WACHSMUTH, S. Towards automated execution and evaluation of simulated prototype HRI experiments. In: **Proceedings of the ACM/IEEE International Conference on Human-robot Interaction**. New York, NY, USA: ACM, 2014. (HRI), p. 230–231. Citado nas páginas 58, 62, 63, 69, 76, 80 e 124.

LILL, R.; SAGLIETTI, F. Testing the cooperation of autonomous robotic agents. In: **9th International Conference on Software Engineering and Applications, ICSOFT-EA**. [S.l.: s.n.], 2014. p. 287–296. Citado nas páginas 58, 62, 63 e 73.

LIM, J.-H.; SONG, S.-H.; KUC, T.-Y.; PARK, H.-S.; KIM, H.-S. A hierarchical test model and automated test framework for RTC. In: LEE, Y.-h.; KIM, T.-h.; FANG, W.-c.; ŚLEZAK, D. (Ed.). **Future Generation Information Technology: First International Conference, FGIT, Jeju Island, Korea**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 198–207. Citado na página 63.

LIMA, L.; CAVALLI, A. A pragmatic approach to generating test sequences for embedded systems. In: **International Federation for Information Processing**. [S.l.: s.n.], 1997. p. 288–307. Citado nas páginas 58, 62, 63, 64 e 84.

LINKMAN, S.; VINCENZI, A. M. R.; MALDONADO, J. An evaluation of systematic functional testing using mutation testing. In: **7th International Conference on Empirical Assessment in Software Engineering - EASE**. Keele, UK: [s.n.], 2003. p. 1–15. Citado nas páginas 37 e 89.

LINNENKUGEL, U.; MÜLLERBURG, M. Test data selection criteria for (software) integration testing. In: **Int. conference on systems integration on Systems integration**. [S.l.: s.n.], 1990. p. 709–717. Citado na página 41.

LU, X.-L.; DONG, Y.-W.; SUN, B.; ZHAO, H.-B. Research of embedded software testing method based on AADL modes. In: **3rd International Conference on Communication Software and Networks**. [S.l.: s.n.], 2011. p. 89–92. Citado nas páginas 58, 62, 63, 68 e 84.

LUCAS, C.; ELBAUM, S.; ROSENBLUM, D. S. Detecting problematic message sequences and frequencies in distributed systems. In: **Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications**. New York, NY, USA: ACM, 2012. (OOPSLA '12), p. 915–926. Citado nas páginas 58, 62, 63, 69, 76, 80, 90, 92 e 124.

LUSSIER, B.; CHATILA, R.; INGRAND, F.; KILLIJIAN, M.; POWELL, D. On fault tolerance and robustness in autonomous systems. In: **3rdI ARP-IEEE/RASEURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments**. [S.l.: s.n.], 2004. Citado nas páginas 82, 83 e 123.

MAIBAUM, T.; LI, Z. J. A test framework for integration testing of object-oriented programs. In: **Conference of the center for advanced studies on Collaborative research**. Riverton, NJ, USA: [s.n.], 2007. p. 252–255. Citado na página 44.

MALDONADO, J. C. **Crítérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software**. Tese (Tese de Doutorado) — DCA/FEE/UNICAMP, Campinas, SP, 1991. Citado nas páginas 29, 36 e 37.

MARWEDEL, P. **Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems**. second. [S.l.]: Springer, 2010. Citado na página 48.

MATHUR, A. P. **Foundations of Software Testing**. [S.l.]: Addison-Wesley Professional, 2008. Citado na página 38.

MATIAS, L. P. N.; SANTOS, T. C.; WOLF, D. F.; SOUZA, J. R. Path planning and autonomous navigation using amcl and AD*. In: **12th Latin American Robotics Symposium and 3rd Brazilian Symposium on Robotics (LARS-SBR)**. [S.l.: s.n.], 2015. p. 320–324. Citado na página 100.

MATTIELLO-FRANCISCO, F. d.; MARTINS, E.; CAVALLI, A.; YANO, E. InRob: An approach for testing interoperability and robustness of real-time embedded software. **Journal of Systems and Software**, v. 85, n. 1, p. 3–15, 2012. Citado nas páginas 58, 59, 61, 62, 63 e 69.

MCCABE, T. A complexity measure. **Transactions on Software Engineering**, 2 (4), pp. 308–320, v. 2, n. 4, p. 308–320, 1976. Citado na página 37.

MELO, S. M.; SOUZA, S. R. S.; SOUZA, P. S. L. Structural testing for multithreaded programs: An experimental evaluation of the cost, strength and effectiveness. In: **International Conference on Software Engineering and Knowledge Engineering**. San Francisco Bay, USA: [s.n.], 2012. p. 476–479. Citado nas páginas 45 e 46.

MILLER, R. E. Comparison of some theoretical models of parallel computation. **IEEE Transactions on Computers**, C-22, n. 8, p. 710–717, 1973. Citado na página 45.

MOLYNEAUX, I. **The Art of Application Performance Testing: Help for Programmers and Quality Assurance**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2009. ISBN 0596520662, 9780596520663. Citado na página 71.

MUNOZ, P.; CESTA, A.; ORLANDINI, A.; R-MORENO, M. D. Evaluating autonomous controllers: An initial assessment. In: **CEUR Workshop Proceedings**. Ferrara, Italy: [s.n.], 2015. v. 1493, p. 1 – 16. ISSN 16130073. Citado nas páginas 58, 62, 63 e 71.

MYERS, G. J. **The Art of Software Testing**. New York: John Wiley & Sons, 1979. Citado nas páginas 29 e 37.

MYERS, G. J.; SANDLER, C.; BADGETT, T.; THOMAS, T. M. **The Art of Software Testing**. [S.l.]: John Wiley & Sons, Inc., Hoboken, New Jersey, 2004. Citado nas páginas 29, 35, 37, 88 e 89.

NEVES, V.; DELAMARO, M.; MASIERO, P. An environment to support structural testing of autonomous vehicles. In: **Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.: s.n.], 2014. p. 19–24. Citado na página 43.

NEVES, V. O. **Automatização do teste estrutural de software de veículos autônomos para apoio ao teste de campo**. Tese (Doutorado), São Carlos, SP, Brasil, 2015. Citado nas páginas 124 e 130.

NEVES, V. O.; LEMOS, O. A. L.; MASIERO, P. Teste estrutural de integração nível 1 de programas orientados a objetos e a aspectos. In: **III Latin American Workshop on Aspect-Oriented Software Development**. [S.l.: s.n.], 2011. Citado na página 43.

OSTRAND, T. J.; BALCER, M. J. The category-partition method for specifying and generating functional tests. **Commun. ACM**, ACM Press, New York, NY, USA, v. 31, n. 6, p. 676–686, 1988. ISSN 0001-0782. Citado na página 37.

PAKALA, H.; KHAN, I.; RAJU, K. Sensors integration in embedded systems. In: **International Conference on Power, Control and Embedded Systems**. [S.l.: s.n.], 2010. Citado nas páginas 58, 62, 63, 68 e 84.

PAKALA, H. G. M.; RAJU, K. V. S. V. N.; KHAN, I. Integration testing of multiple embedded processing components. **Advanced Computing**, v. 133, p. 200–209, 2011. Citado nas páginas 58, 62, 63, 65 e 84.

PARK, M.-Y.; SHIM, S. J.; JUN, Y.-K.; PARK, H.-R. MPIRace-check: detection of message races in MPI programs. In: **2nd international conference on Advances in grid and pervasive computing**. [S.l.: s.n.], 2007. p. 322–333. Citado na página 45.

PéREZ, A.; KAISER, S. Reusing component test cases for integration testing of retarding embedded system components. In: **First International Conference on Advances in System Testing and Validation Lifecycle**. [S.l.: s.n.], 2009. p. 1–6. Citado nas páginas 58, 62, 63 e 67.

PéREZ, M.; KAISER, S. Bottom-up reuse for multi-level testing. **Journal of Systems and Software**, v. 83, n. 12, p. 2392–2415, 2010. Citado nas páginas 58, 62, 63 e 67.

PERRY, D. E.; KAISER, G. E. Adequate testing and object-oriented programming. **J. Object Oriented Program.**, SIGS Publications, Denville, NJ, USA, v. 2, n. 5, p. 13–19, jan. 1990. Citado na página 42.

PESONEN, J. Extending software integration testing using aspects in symbian os. In: **Testing: Academic and Industrial Conference - Practice and Research Techniques**. [S.l.: s.n.], 2006. p. 147–151. Citado nas páginas 58, 62, 63 e 66.

PETTERS, S.; THOMAS, D.; FRIEDMANN, M.; STRYK, O. V. Multilevel testing of control software for teams of autonomous mobile robots. In: CARPIN, S.; NODA, I.; PAGELLO, E.; REGGIANI, M.; STRYK, O. von (Ed.). **Simulation, Modeling, and Programming for Autonomous Robots: First International Conference, SIMPAR**. [S.l.: s.n.], 2008. v. 5325 LNAI, p. 183–194. Citado nas páginas 58, 62, 63 e 67.

PETTERS, S.; THOMAS, D.; STRYK, O. V. RoboFrame- a modular software framework for lightweight autonomous robots. In: **Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems**. San Diego, CA, USA: [s.n.], 2007. Citado na página 67.

PLAKU, E.; KAVRAKI, L. E.; VARDI, M. Y. Hybrid systems: From verification to falsification. In: **Computer Aided Verification: 19th International Conference, CAV, Berlin, Germany, July 3-7, Proceedings**. Berlin, Germany: Springer Berlin Heidelberg, 2007. v. 4590 LNCS, p. 463 – 476. ISSN 03029743. Citado nas páginas 58, 62, 63 e 64.

POWELL, D.; ARLAT, J.; CHU, H.; INGRAND, F.; KILLIJIAN, M.-O. Testing the input timing robustness of real-time control software for autonomous systems. In: **9th European Dependable Computing Conference, EDCC**. [S.l.: s.n.], 2012. p. 73–83. Citado nas páginas 58, 62, 63 e 72.

PRESSMAN, R. S. **Software Engineering - A Practitioner's Approach**. 5^a. ed. [S.l.]: McGraw-Hill, 2000. Citado na página 37.

_____. Software engineering: Practitioner's approach. In: _____. 6. ed. [S.l.]: McGraw-Hill, 2005. Citado nas páginas 36, 37 e 41.

PU, F.; XU, H.-Y. A feasible strategy for reachability testing of internet-based concurrent programs. In: **IEEE International Conference on Networking, Sensing and Control**. [S.l.: s.n.], 2008. p. 1559–1564. Citado na página 45.

QUIGLEY, M.; CONLEY, K.; GERKEY, B. P.; FAUST, J.; FOOTE, T.; LEIBS, J.; WHEELER, R.; NG, A. Y. ROS: an open-source robot operating system. In: **ICRA Workshop on Open Source Software**. [S.l.: s.n.], 2009. Citado nas páginas 48 e 50.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. **IEEE Transactions on Software Engineering**, SE-11, n. 4, p. 367–375, April 1985. ISSN 0098-5589. Citado nas páginas 29, 37 e 85.

RIBEIRO, M. I. **Localização em Robótica Móvel - Odometria**. 1999. Acesso em: 20.Dez.2016. Disponível em: <<http://users.isr.ist.utl.pt/~mir/cadeiras/robmovel/Odometry.pdf>>. Citado nas páginas 82 e 123.

ROSENBLUM, D. S. **Adequate testing of component-based software**. Irvine, CA, EUA, 1997. University of California, UCI-ICS-97-34. Citado na página 43.

ROSHAN, R.; PORWAL, R.; SHARMA, C. Review of search based techniques in software testing. **International Journal of Computer Applications**, v. 51, n. 6, p. 42–45, 2012. ISSN 0975-8887. Citado na página 73.

RUBINOV, K. Generating integration test cases automatically. In: **International symposium on Foundations of software engineering**. New York, NY, USA: ACM, 2010. p. 357–360. Citado na página 44.

SAGLIETTI, F.; PINTE, F. Automated unit and integration testing for component-based software systems. In: **International Workshop on Security and Dependability for Resource Constrained Embedded Systems**. [S.l.: s.n.], 2010. Citado nas páginas 58, 59, 61, 62, 63 e 73.

SARMANHO, F. S.; SOUZA, P. S.; SOUZA, S. R.; SIMÃO, A. S. Structural testing for semaphore-based multithread programs. In: **8th international conference on Computational Science**. Berlin, Heidelberg: Springer-Verlag, 2008. p. 337–346. Citado nas páginas 31, 45 e 46.

SASNAUSKAS, R.; LANDSIEDEL, O.; ALIZAI, M. H.; WEISE, C.; KOWALEWSKI, S.; WEHRLE, K. Kleenet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In: **Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks**. [S.l.: s.n.], 2010. (IPSN '10), p. 186–196. ISBN 978-1-60558-988-6. Citado na página 83.

SATOH, I. A software testing framework for networked industrial systems. In: **IECON Proceedings (Industrial Electronics Conference)**. [S.l.: s.n.], 2013. p. 4340–4345. Citado nas páginas 58, 62, 63 e 72.

SCHATZ, B.; PFALLER, C. Integrating component tests to system tests. **Electronic Notes in Theoretical Computer Science**, v. 260, p. 225–241, 2010. Citado nas páginas 58, 62, 63 e 74.

- SCHERER, S.; LERDA, F.; CLARKE, E. Model checking of robotic control systems. **European Space Agency, (Special Publication) ESA SP**, Munich, Germany, n. 603, p. 371 – 378, 2005. ISSN 03796566. Citado nas páginas 58, 62 e 63.
- SEN, A.; ABADIR, M. Coverage metrics for verification of concurrent systemc designs using mutation testing. In: **IEEE International High Level Design Validation and Test Workshop**. [S.l.: s.n.], 2010. p. 75 –81. Citado na página 45.
- SEO, J.; SUNG, A.; CHOI, B.; KANG, S. Automating embedded software testing on an emulated target board. In: **Second International Workshop on Automation of Software Test**. Washington, DC, USA: [s.n.], 2007. p. 9–. Citado nas páginas 58, 62, 63 e 66.
- SHARYGINA, N.; PELED, D. A combined testing and verification approach for software reliability. In: **International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity**. London, UK, UK: Springer-Verlag, 2001. p. 611–628. ISBN 3-540-41791-5. Citado nas páginas 58, 60, 62, 63 e 65.
- SILVA, R. A.; SOUZA, S. R. S.; SOUZA, P. S. L. Mutation operators for concurrent programs in MPI. In: **IEEE Latin American Test Workshop**. Quito: [s.n.], 2012. p. 69–74. Citado nas páginas 45 e 46.
- SIMÃO, A. Introdução ao teste de software. In: _____. [S.l.]: M. E. Delamaro and J. C. Maldonado and M. Jino, 2007. v. 1, n. 1, cap. Teste Baseado em Modelos, p. 27–45. Citado na página 38.
- SOLHEIM, J. A.; ROWLAND, J. H. An empirical study of testing and integration strategies using artificial software systems. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 19, n. 10, p. 941–949, out. 1993. Citado na página 41.
- SOUZA, P. S. L.; SAWABE, E. T.; SIMÃO, A. S.; VERGILIO, S. R.; SOUZA, S. R. S.; SARMANHO, F. ValiPVM - a graphical tool for structural testing of PVM programs. In: **Euro PVM/MPI - LNCS - Recent Advances in Parallel Virtual Machine and Message Passing Interface**. Dublin, Ireland: [s.n.], 2008. p. 257–264. Citado nas páginas 45 e 46.
- SOUZA, P. S. L.; SOUZA, S. R. S.; ZALUSKA, E. Structural testing for message-passing concurrent programs: an extended test model. **Concurrency and Computation: Practice and Experience**, v. 26, n. 1, p. 21–50, 2014. Citado nas páginas 32, 45, 46, 77 e 86.
- SOUZA, S. R. S.; SOUZA, P. S. L.; MACHADO, M. C. C.; CAMILLO, M. S.; SIMÃO, A. S.; ZALUSKA, E. Using coverage and reachability testing to improve concurrent program testing quality. In: **Int. Conf. on Software Engineering and Knowledge Engineering**. Eden Roc Renaissance Miami Beach, USA: [s.n.], 2011. p. 207–212. Citado nas páginas 31, 32, 45 e 46.
- SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, P. S. L. Introdução ao teste de software. In: _____. [S.l.]: M. E. Delamaro and J. C. Maldonado and M. Jino, 2007. v. 1, n. 1, cap. Teste de Programas Concorrentes, p. 231–249. Citado na página 45.
- SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, P. S. L.; SIMÃO, A. S.; HAUSEN, A. C. Structural testing criteria for message-passing parallel programs. **Concurrency and Computation. Practice and Experience**, John Wiley and Sons Ltd., Chichester, UK, v. 20, n. 16, p. 1893–1916, nov. 2008. Citado nas páginas 31, 32, 45, 46, 77 e 86.
- SUNG, A.; CHOI, B.; SHIN, S. An interface test model for hardware-dependent software and embedded OS API of the embedded system. **Computer Standards and Interfaces**, v. 29, n. 4, p. 430–443, 2007. Citado nas páginas 58, 62, 63 e 74.

SUNG, A.; CHOI, B.; WONG, W.; DEBROY, V. Mutant generation for embedded systems using kernel-based software and hardware fault simulation. **Information and Software Technology**, v. 53, n. 10, p. 1153–1164, 2011. Citado nas páginas 58, 62, 63 e 68.

SZATMÁRI, Z.; OLÁH, J.; MAJZIK, I. Ontology-based test data generation using metaheuristics. In: **8th International Conference on Informatics in Control, Automation and Robotics**. [S.l.: s.n.], 2011. v. 2, p. 217–222. Citado nas páginas 58, 62, 63 e 73.

SZYPERSKI, C. **Component Software: Beyond Object-Oriented Programming**. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0201745720. Citado nas páginas 43 e 65.

TANENBAUM, A. S. **Modern operating systems**. Second. [S.l.: s.n.], 2001. xxiv + 951 p. ISBN 0-13-031358-0. Citado nas páginas 44, 78 e 80.

TAYLOR, R.; LEVINE, D.; KELLY, C. Structural testing of concurrent programs. **IEEE Transactions on Software Engineering**, v. 18, n. 3, p. 206–215, Mar 1992. Citado na página 45.

TAYLOR, R. N. A general-purpose algorithm for analyzing concurrent programs. **Commun. ACM**, ACM, New York, NY, USA, v. 26, n. 5, p. 361–376, 1983. ISSN 0001-0782. Citado na página 45.

THRUN, S.; BURGARD, W.; FOX, D. **Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)**. [S.l.]: The MIT Press, 2005. Citado na página 29.

THRUN, S.; MONTEMERLO, M.; DAHLKAMP, H.; STAVENS, D.; ARON, A.; DIEBEL, J.; FONG, P.; GALE, J.; HALPENNY, M.; HOFFMANN, G.; LAU, K.; OAKLEY, C.; PALATUCCI, M.; PRATT, V.; STANG, P.; STROHBAND, S.; DUPONT, C.; JENDROSSEK, L.-E.; KOELEN, C.; MARKEY, C.; RUMMEL, C.; NIEKERK, J. van; JENSEN, E.; ALESSANDRINI, P.; BRADSKI, G.; DAVIES, B.; ETTINGER, S.; KAEHLER, A.; NEFIAN, A.; MAHONEY, P. Stanley: The robot that won the DARPA grand challenge: Research articles. **Journal of Field Robotics**, John Wiley and Sons Ltd., Chichester, UK, v. 23, n. 9, p. 661–692, set. 2006. ISSN 0741-2223. Citado na página 47.

TSAI, C.-S.; HSIEH, C.-H. Development of a computer integrated testing system for humanoid robots ex-work inspection. **Applied Mechanics and Materials**, v. 190-191, p. 683–687, 2012. Citado nas páginas 58, 62, 63 e 75.

TYLER, B.; SOUNDARAJAN, N. Testing framework components. In: **Component-Based Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2004. v. 3054, p. 138–145. Citado na página 43.

URAL, H.; YANG, B. A structural test selection criterion. **Inf. Process. Lett.**, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 28, n. 3, p. 157–163, 1988. ISSN 0020-0190. Citado na página 29.

VINCENZI, A. M. R.; DELAMARO, M. E.; MALDONADO, J. C.; WONG, W. E. Establishing structural testing criteria for Java bytecode. **Software Practice and Experience**, John Wiley & Sons, Inc., New York, NY, USA, v. 36, n. 14, p. 1513–1541, nov. 2006. ISSN 0038-0644. Citado na página 42.

- VINCENZI, A. M. R.; MALDONADO, J. C.; BARBOSA, E. F.; DELAMARO, M. E. Operadores essenciais de interface: Um estudo de caso. In: **XIII Simpósio Brasileiro de Engenharia de Software**. Florianópolis, SC, Brasil: [s.n.], 1999. p. 373–391. Citado na página 42.
- WEISSHARDT, F.; KETT, J.; ARAUJO, T. D. F. O.; BUBECK, A.; VERL, A. Enhancing software portability with a testing and evaluation platform. In: **Proceedings for the Joint Conference of ISR 2014 - 45th International Symposium on Robotics and Robotik - 8th German Conference on Robotics, ISR/ROBOTIK**. Munich, Germany: [s.n.], 2014. p. 219 – 224. Citado nas páginas 58, 62, 63, 69, 76, 80 e 124.
- WOLF, D.; SIMÕES, E.; OSÓRIO, F.; JÚNIOR, O. T. **Robótica Móvel Inteligente: Da Simulação às Aplicações no Mundo Real**. Bento Gonçalves/RS, 2009. Citado nas páginas 48, 49, 50 e 52.
- WONG W.E., L. Y. Reachability graph-based test sequence generation for concurrent programs. **International Journal of Software Engineering and Knowledge Engineering**, v. 18, n. 6, p. 803–822, 2008. Citado na página 45.
- WU, Y.; CHEN, M.-H.; OFFUTT, J. UML-based integration testing for component-based software. In: **Second International Conference on COTS-Based Software Systems**. London, UK, UK: Springer-Verlag, 2003. p. 251–260. Citado na página 44.
- YANG, C.-S. D.; SOUTER, A. L.; POLLOCK, L. L. All-du-path coverage for parallel programs. In: **international symposium on Software testing and analysis**. New York, NY, USA: [s.n.], 1998. p. 153–162. ISBN 0-89791-971-8. Citado na página 45.
- YANG, R.-D.; CHUNG, C.-G. Path analysis testing of concurrent programs. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 34, n. 1, p. 43–56, 1992. ISSN 0950-5849. Citado na página 45.
- YOON, J. H.; KANG, J. S.; PARK, H. S. Test data combination strategy for effective test suite generation. In: **International Conference on IT Convergence and Security, ICITCS**. Macau, China: [s.n.], 2013. Citado nas páginas 58, 62, 63 e 74.
- YOUNG, M.; TAYLOR, R. N.; LEVINE, D. L.; NIES, K. A.; BRODBECK, D. **A Concurrency Analysis Tool Suite for Ada Programs: Rationale, Design, and Preliminary Experience**. New York, NY, USA, 1995. v. 4, n. 1, 65–106 p. Citado na página 45.
- YUAN, H.; XIE, T. Substra: a framework for automatic generation of integration tests. In: **International workshop on Automation of software test**. New York, NY, USA: [s.n.], 2006. p. 64–70. Citado na página 44.

CÓDIGO FONTE SISTEMA DO ROBÔ ASPIRADOR DE PÓ

Código-fonte 1 – Código fonte dos processos Collision Avoidance (processo 2) e Controller (processo 4)

```

1: /*****/
2: /*    Programa Aspirador de po    */
3: /*    processo 2                    */
4: /*****/
5: /*01*/void proximityCallback(const std_msgs::Int8::ConstPtr &
    proximityP){
6:
7: /*01*/    proximity = proximityP->data;
8: /*01*/}
9:
10: /*01*/void velocityCallback(const std_msgs::Int8::ConstPtr &
    speedP){
11:
12: /*01*/    if((speedP -> data) == 0){
13: /*02*/        time_to_impact = 100;
14: /*02*/    }
15: /*03*/    else{
16: /*03*/        time_to_impact = (proximity/(speedP->data));
17: /*03*/    }
18: /*04*/    if(time_to_impact < 2) {
19: /*05*/        if(state == working){
20: /*06*/            state = crashed;
21: /*06*/            assert(false); // Failure

```

```
22: /*06*/      }
23: /*07*/      else{
24: /*07*/          state = stopped;
25: /*07*/      }
26: /*07*/  }
27: /*08*/  else
28: /*08*/      if(time_to_impact < 3) {
29: /*09*/          state = slowed;
30: /*09*/          ponto.reduceMapping();
31: /*09*/      }
32: /*10*/  else{
33: /*10*/      state = working;
34: /*10*/      ponto.activateMapping();
35: /*10*/  }
36: /*11*/}
37:
38: /*01*/int main(int argc, char * * argv){
39:
40: /*01*/  ros::init(argc, argv, "Collision_Avoidance");
41: /*01*/  ROS_INFO("Collision...");
42: /*01*/  ros::NodeHandle n;
43: /*01*/  ros::Rate loop_rate(0.6);
44: /*01*/  ros::Subscriber sub1 = n.subscribe("velocity", 1,
      velocityCallback);
45: /*01*/  ros::Subscriber sub = n.subscribe("proximity", 1,
      proximityCallback);
46: /*01*/  ros::Publisher chatter_pub = n.advertise < std_msgs::
      Int8 > ("timeToImpact", 1);
47: /*01*/  ros::Publisher chatter_pub1 = n.advertise < std_msgs::
      Int8 > ("mapper_activation", 1);
48: /*01*/  std_msgs::Int8 msgTimeToImpact;
49: /*01*/  std_msgs::Int8 msgMapperActivation;
50: /*02*/  while (ros::ok()) {
51: /*03*/      msgTimeToImpact.data = time_to_impact;
52: /*03*/      msgMapperActivation.data = (ponto.getW());
53: /*03*/      chatter_pub.publish(msgTimeToImpact);
54: /*03*/      chatter_pub1.publish(msgMapperActivation);
55: /*03*/      ros::spinOnce();
56: /*03*/      loop_rate.sleep();
57: /*03*/  }
58: /*04*/  return 0;
59: /*04*/}
```

```

60:
61: /*****
62:  /*      Programa Aspirador de po      */
63:  /*      processo 4                      */
64:  *****/
65: /*01*/void odometryCallback(const std_msgs::Int8::ConstPtr &
        odometryP){
66:
67: /*01*/  odometry = odometryP->data;
68: /*01*/}
69:
70: /*01*/void timeToImpactCallback(const std_msgs::Int8::ConstPtr
        & time_impactP){
71:
72: /*01*/  time_to_impact = (time_impactP->data);
73: /*01*/  if((velocity == (time_to_impact - 1))){
74: /*02*/    statePoint = 1;
75: /*02*/    }
76: /*03*/  else
77: /*03*/    if((time_to_impact > 3)){
78: /*04*/      velocity = 3;
79: /*04*/      cmd = 3;
80: /*04*/      statePoint = 1;
81: /*04*/    }
82: /*05*/  else
83: /*05*/    if((time_to_impact == 3) && (velocity == 3)){
84: /*06*/      velocity = 3;
85: /*06*/      cmd = 3;
86: /*06*/      statePoint = 1;
87: /*06*/    }
88: /*07*/  else
89: /*07*/    if(((time_to_impact == 2 ) && ((velocity == 3) || (
        velocity == 2)))){
90: /*08*/      velocity = 2;
91: /*08*/      cmd = 2;
92: /*08*/      statePoint = 1;
93: /*08*/    }
94: /*09*/  else
95: /*09*/    if((time_to_impact == 1 ) && (velocity == 2)){
96: /*10*/      velocity = 1;
97: /*10*/      cmd = 1;
98: /*10*/      statePoint = 0;

```

```

99: /*10*/ }
100: /*11*/ else
101: /*11*/     if((time_to_impact == 1)){
102: /*12*/         velocity = 1;
103: /*12*/         cmd = 1;
104: /*12*/         statePoint = 0;
105: /*12*/     }
106: /*13*/ else
107: /*13*/     if((time_to_impact == 0) && (velocity == 1)){
108: /*14*/         velocity = 1;
109: /*14*/         cmd = 1;
110: /*14*/         statePoint = 0;
111: /*14*/     }
112: /*15*/}
113:
114: /*01*/int main(int argc, char * * argv){
115:
116: /*01*/     ros::init(argc, argv, "Controller");
117: /*01*/     ROS_INFO("Controller...");
118: /*01*/     ros::NodeHandle n;
119: /*01*/     std_msgs::Int8 msgCmdMotors;
120: /*01*/     std_msgs::Int8 msgVelocity;
121: /*01*/     geometry_msgs::Point msgLocation;
122: /*01*/     ros::Publisher chatter_pub = n.advertise < std_msgs::
        Int8 > ("velocity", 1);
123: /*01*/     ros::Publisher chatter_pub1 = n.advertise < std_msgs::
        Int8 > ("cmd_motors", 1);
124: /*01*/     ros::Publisher chatter_pub2 = n.advertise <
        geometry_msgs::Point> ("location", 1);
125: /*01*/     ros::Subscriber sub = n.subscribe("timeToImpact", 1,
        timeToImpactCallback);
126: /*01*/     ros::Subscriber sub1 = n.subscribe("odometry", 1,
        odometryCallback);
127: /*01*/     ros::Rate loop_rate(0.6);
128: /*01*/     (msgVelocity.data) = velocity;
129: /*02*/     while (ros::ok()) {
130: /*03*/         msgCmdMotors.data = cmd;
131: /*03*/         ponto.setStatePoint(statePoint);
132: /*03*/         msgLocation.x = ponto.x;
133: /*03*/         msgLocation.y = ponto.y;
134: /*03*/         msgLocation.z = ponto.z;
135: /*03*/         chatter_pub.publish(msgVelocity);

```

```
136: /*03*/          chatter_pub1.publish(msgCmdMotors);
137: /*03*/          chatter_pub2.publish(msgLocation);
138: /*03*/          ros::spinOnce();
139: /*03*/          loop_rate.sleep();
140: /*03*/      }
141: /*04*/      return 0;
142: /*04*/}
```