Estudo sobre adaptação de critérios de teste de programas concorrentes para o teste de integração de sistemas robóticos móveis

Marcos Pereira dos Santos

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Marcos Pereira dos Santos

Estudo sobre adaptação de critérios de teste de programas concorrentes para o teste de integração de sistemas robóticos móveis

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre 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 Souza

USP – São Carlos
Junho de 2016

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

S634e

Santos, Marcos Pereira dos
Estudo sobre adaptação de critérios de teste de programas concorrentes para o teste de integração de sistemas robóticos móveis / Marcos Pereira dos Santos; orientadora Simone do Rocio Senger Souza. - São Carlos - SP, 2016.
86 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2016.

1. teste de programas concorrentes. 2. critérios de teste. 3. sistemas robóticos móveis. I. Souza, Simone do Rocio Senger, orient. II. Título.

Marcos Pereira dos Santos

Study about adaptation of the concurrent programs testing criteria for integration testing of mobile robotic systems

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

Concentration Area: Computer Science and Computational Mathematics

Advisor: Profa. Dra. Simone do Rocio Senger Souza

USP – São Carlos
June 2016

Dedico esta dissertação a todos que acompanharam e torceram pela minha jornada.

AGRADECIMENTOS

Agradeço

Em primeiro lugar a Deus, pelo dom da vida e por me dar saúde, paz e todas as condições para me conduzir até aqui.

Especialmente aos meus pais, o Terlindo e a Geralda, pela educação e apoio na minha caminhada.

Aos meus irmãos: Renata, Marcelo e Maiara.

A professora Simone que com muita sabedoria, paciência e muita dedicação me orientou no desenvolvimento deste trabalho. Ao professor Paulo Sérgio pela ajuda e opnião sempre importantes.

À Maria pelo auxílio e pelas discussões no desenvolvimento do trabalho.

À todos do Labes, do Apostaqui, da Bem Amigos, da Eita Caralha que fizeram com que minha estadia em São Carlos se tornasse melhor.

Aos alunos e professores do LRM pela disponibilidade e pelo auxílio quando requisitado.

À todos os meus amigos e familiares que torceram e me apoiaram na batalha para realização deste trabalho.

À todos que de uma forma ou de outra contribuíram para que eu pudesse chegar até aqui.

Por fim, agradeço ao Capes pelo apoio financeiro.

"Os meus heróis são aqueles que sobreviveram
ao fazer algo errado, que cometeram erros,
mas que conseguiram ultrapassá-los e corrigi-los."
(Bono Vox)

RESUMO

SANTOS, M. P.. **Estudo sobre adaptação de critérios de teste de programas concorrentes para o teste de integração de sistemas robóticos móveis**. 2016. 86 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

A definição de abordagens de teste de software no contexto de sistemas embarcados críticos é de fundamental importância para a melhoria de qualidade desses sistemas. Este projeto de mestrado concentra-se essencialmente na área de robôs móveis que são sistemas críticos. O mapeamento das técnicas de teste dos programas tradicionais para sistemas robóticos móveis não é trivial, pois é necessário considerar as características inerentes a esses sistemas, as quais incluem comunicação, sincronização, não determinismo e paralelismo. Esses sistemas são formados por diferentes componentes, como sensores, atuadores e softwares de controle, os quais se comunicam, em geral, por meio de troca de mensagens. Nesse sentido, este projeto visa aproveitar a experiência adquirida pelo grupo de pesquisa do ICMC/USP com a proposição de critérios e ferramentas de teste para programas concorrentes, para o contexto de teste de integração de sistemas robóticos móveis. Assim, neste estudo é apresentado um comparativo sobre as similaridades e diferenças presentes em programas concorrentes e sistemas robóticos móveis com objetivo de auxiliar na adaptação dos critérios de teste. Dois estudos de caso foram conduzidos nos quais os critérios propostos para programas concorrentes foram aplicados a sistemas robóticos desenvolvidos em ROS. Como resultados, observou-se os critérios de teste são capazes de testar adequadamente aspectos de comunicação e sincronização de sistemas robóticos, contribuindo com a qualidade desses sistemas.

Palavras-chave: teste de programas concorrentes, critérios de teste, sistemas robóticos móveis.

ABSTRACT

SANTOS, M. P.. **Estudo sobre adaptação de critérios de teste de programas concorrentes para o teste de integração de sistemas robóticos móveis**. 2016. 86 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

The definition of software testing approaches for the context of critical embedded systems is of fundamental importance for the quality improvement of these systems. This master's project focuses primarily in mobile robot area that are critical systems. The mapping of traditional testing techniques for robotic mobile systems is not trivial, because it is necessary to consider the characteristics of these systems, which include communication, synchronization, non-determinism and parallelism. These systems present different components such as sensors, actuators and control software, which communicate, usually by exchanging messages. In this sense, this project aims to use on the experience gained by the research group of the ICMC/USP with the of the concurrent program testing criteria for the integration testing of the mobile robotic systems. In this study we present a comparison of the similarities and differences present in concurrent programs and mobile robotic systems in order to assist in adapting the testing criteria. Two case studies were conducted in which the proposed concurrent program testing criteri were applied to robotic systems developed in ROS. As a result, there was the testing criteria are able to properly testing aspects of communication and synchronization of robotic systems, contributing to the quality of these systems.

Key-words: concurrent program testing, testing criteria, mobile robotic system.

LISTA DE ILUSTRAÇÕES

Figura 1 – Tipos de erros de integração: (a) Erro Tipo 1, (b) Erro Tipo 2 e (c) Erro Tipo 3 (DELAMARO, 1997). 30
Figura 2 – Exemplo de robôs móveis: CaRINA veículo Palio Adventure. 36
Figura 3 – Organização do sistema de arquivo do ROS (FERNANDES, 2012). 40
Figura 4 – Grafo de visualização de um sistema ROS gerado pela ferramenta *rxgraph*. . 40
Figura 5 – Níveis de teste no ROS 44
Figura 6 – Visão geral do modelo de teste (LIM *et al.*, 2009) 51
Figura 7 – Grafo gerado pelo ROS para o robô aspirador. 59
Figura 8 – GFCP para o robô aspirador. 62
Figura 9 – Exemplo de configuração do Rostest para execução dos testes. 66
Figura 10 – CaRINA Parking (MATIAS *et al.*, 2015) 72
Figura 11 – Grafo gerado pelo ROS para a aplicação do Carina Parking. 72
Figura 12 – GFCP para o Carina Parking. 73

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Exemplo de código de um publisher() 41
Código-fonte 2 – Exemplo de código de um subscriber() 41
Código-fonte 3 – Exemplo de configuração de um ROSTEST. 42
Código-fonte 4 – Exemplo de configuração de código Gtest. 43
Código-fonte 5 – Resumo do código do componente *Collision Avoidance* (LUCAS; ELBAUM; ROSENBLUM, 2012) component. 59
Código-fonte 6 – Resumo do código do componente *Controller* 60
Código-fonte 7 – Exemplo de caso de teste executado. 67

LISTA DE TABELAS

Tabela 1 – Mapeamento dos conceitos do modelo de teste de programas concorrentes para sistemas robóticos. 62
Tabela 2 – Informações obtidas pelo modelo de teste para o robô aspirador de pó. . . . 63
Tabela 3 – Variáveis definidas para o robô aspirador de pó. 64
Tabela 4 – Elementos requeridos para os critérios para o robô aspirador de pó. 65
Tabela 5 – Casos de teste. 66
Tabela 6 – Informações obtidas pelo modelo de teste para o robô Carina Parking. . . . 74
Tabela 7 – Alguns elementos requeridos pelos critérios para o robô Carina Parking. . . 75

SUMÁRIO

1 INTRODUÇÃO 23
1.1 Contextualização 23
1.2 Motivação 25
1.3 Objetivos 26
1.4 Estrutura 26

2 FUNDAMENTAÇÃO TEÓRICA 27
2.1 Considerações Iniciais 27
2.2 Teste de Software 28
2.2.1 Técnica Estrutural 32
2.3 Sistemas Robóticos Móveis 34
2.3.1 Metodologias para Desenvolvimento de Robôs Móveis 36
2.3.2 Robotic Operating System - ROS 38
2.3.2.1 Paradigma Publisher/Subscriber 41
2.3.2.2 Ambientes de Testes no ROS 42
2.4 Teste Estrutural de Programas Concorrentes 44
2.5 Considerações Finais 48

3 TRABALHOS RELACIONADOS 49
3.1 Considerações Iniciais 49
3.2 Contextualização 50
3.3 Teste de Software Aplicado a Sistemas Embarcados Robóticos . . . 50
3.4 Considerações Finais 53

4 TESTE DE PROGRAMAS CONCORRENTES APLICADO A SISTEMAS ROBÓTICOS MÓVEIS 55
4.1 Considerações Iniciais 55
4.2 Mapeamento dos Conceitos do Modelo de Teste 56
4.3 Aplicação dos critérios de teste 58
4.3.1 Estudo de caso: Robô iRobot Roomba 58
4.3.2 Aplicação dos critérios de teste para programas concorrentes 60
4.3.2.1 Ferramentas de apoio a execução dos testes 64
4.4 Discussões 69
4.5 Considerações Finais 69

5 ESTUDO DE CASO 2 71
5.1 Considerações Iniciais 71
5.2 A Aplicação 71
5.3 Aplicação dos Critérios de Teste de Programas Concorrentes 73
5.3.1 Execução dos Testes *74*
5.4 Discussões e Análise dos Resultados 76
5.5 Considerações Finais 76

6 CONCLUSÕES 77
6.1 Caracterização da Pesquisa Realizada 77
6.2 Contribuições 77
6.3 Desafios, Dificuldades e Limitações 78
6.4 Trabalhos Futuros 79

REFERÊNCIAS 81

CAPÍTULO

1

INTRODUÇÃO

1.1 Contextualização

Dentre os mecanismos propostos pela Engenharia de Software para auxiliar no desenvolvimento de software, as atividades de VV&T (Validação, Verificação e Teste) visam garantir a qualidade, confiabilidade e segurança do software produzido (DELAMARO; MALDONADO; JINO, 2007).

A atividade de teste se constitui como uma análise dinâmica do produto e tem por objetivo encontrar os defeitos presentes no mesmo. Para isso, técnicas e critérios de teste são propostos, os quais devem ser aplicados nas diferentes fases do processo de desenvolvimento (MYERS, 2004; DELAMARO; MALDONADO; JINO, 2007).

O teste de produtos de software envolve basicamente quatro etapas: planejamento de testes, projeto de casos de teste, execução e avaliação dos resultados dos testes. Essas atividades devem ser desenvolvidas ao longo do processo de desenvolvimento de software. O conjunto de informação obtido da atividade de teste é significativo para as atividades de depuração, manutenção e estimativa de confiabilidade de software (DELAMARO; MALDONADO; JINO, 2007).

No contexto de programas sequenciais foram definidos ao longo dos anos várias técnicas e critérios de teste (HERMAN, 1976; HOWDEN, 1975; DEMILLO; LIPTON; SAYAWARD, 1978; MYERS, 2004; LASKI; KOREL, 1983; NTAFOS, 1984; RAPPS; WEYUKER, 1985; URAL; YANG, 1988; COWARD, 1988; BEIZER, 1990; MALDONADO, 1991). Esses estudos oferecem a base fundamental para a proposição de novos critérios e modelos de teste, considerando diferentes paradigmas de programação, como Orientação a Objetos e Orientação a Aspectos, além de diferentes domínios de aplicação, como sistemas distribuídos, concorrentes, de tempo real e embarcados (DELAMARO; MALDONADO; JINO, 2007).

O teste de aplicações concorrentes comparado ao teste de programas sequenciais é mais

complicado, pois além das dificuldades inerentes à atividade de teste, novos desafios são impostos principalmente devido ao comportamento não determinístico em que diferentes sequências de sincronização podem ser obtidas e gerarem diferentes saídas para uma mesma entrada de teste, e por isso, faz-se necessário avaliar se o comportamento das saídas é o esperado ou não. Neste sentido, os grupos de pesquisa do ICMC têm explorado a definição de mecanismos de apoio para o teste de unidade de programas concorrentes, os quais podem ser divididos em três grupos principais de contribuições: 1) proposição de modelos e critérios de teste; 2) proposição de mecanismos de apoio à execução controlada; e 3) definição de ferramentas de apoio ao teste de programas concorrentes (SOUZA *et al.*, 2005; VERGILIO; SOUZA; SOUZA, 2005; HAUSEN *et al.*, 2007; SOUZA *et al.*, 2008; ENDO *et al.*, 2008; SARMANHO *et al.*, 2008; SOUZA; SOUZA; ZALUSKA, 2012; BRITO; SOUZA; SOUZA, 2013; SOUZA *et al.*, 2013).

A programação concorrente consiste no princípio de decompor a resolução dos problemas em atividades menores e independentes que possam ser executadas em múltiplos processadores. Essas atividades quando executadas concorrentemente tendem a reduzir a ociosidade nos núcleos de processamento melhorando o desempenho das aplicações, explorando a concorrência em diferentes arquiteturas, permitindo assim uma melhor utilização dos recursos disponíveis na resolução do problema (GRAMA *et al.*, 2003a; PACHECO, 2011).

A robótica móvel é a ciência que percebe e manipula o mundo físico por meio de dispositivos computacionais. Nesses sistemas são necessários mecanismos para perceber e atuar no ambiente. Pesquisas são conduzidas para o desenvolvimento de robôs para patrulhamento interno, com aplicações em segurança, transporte, saúde, dentre diversas outras áreas. Nesse domínio de aplicações, destaca-se o desenvolvimento de robôs móveis autônomos, os quais devem ser capazes de interagir com o ambiente e tomar decisões corretas para que suas tarefas sejam executadas com êxito. Para o desenvolvimento de robôs móveis são utilizados ambientes que permitem simular o comportamento dos robôs à medida que estão sendo desenvolvidos. Exemplos destes ambientes são: ROS, Player, Ária, Carmen, Microsoft Robotics Studio, dentre diversos outros (WOLF *et al.*, 2009).

Sistemas para robôs móveis são também categorizados como sistemas embarcados. Um sistema embarcado, também denominado sistema embutido, é formado por um computador encapsulado ou totalmente dedicado ao sistema que ele controla. Pode ser definido como um sistema de computador, com um objetivo específico contendo uma combinação de hardware e software (e opcionalmente mecânica, elétrica, dentre outros) projetado para desempenhar uma função dedicada, interconectada e embutida num produto (BROEKMAN; NOTENBOOM, 2002). Uma das principais características de um sistema embarcado é sua forma de interação com o mundo real, a qual é realizada por meio de sinais de entrada (captados por sensores) e sinais de saída (enviados por atuadores). O teste de sistemas embarcados busca garantir confiabilidade, desempenho e robustez, ao verificar se todas as funcionalidades estão em conformidade com a sua especificação (BALAKRISHNAN; ANAND, 2009).

A maioria dos softwares embarcados tem um ambiente de desenvolvimento que difere do ambiente destino. O ambiente de desenvolvimento, muitas vezes, é dotado de simuladores e outros meios que buscam verificar se o software executa conforme o esperado, não permitindo mais depurar ou alterar o software, como ocorria no ambiente de desenvolvimento (TIAN *et al.*, 2009). Essa característica demonstra a dificuldade em testar/depurar esses sistemas, pois muitos dos problemas, principalmente relacionados à comunicação entre os componentes, podem surgir somente depois que o software está carregado no ambiente final.

Diante disso, algumas características dos sistemas embarcados devem ser observadas durante a proposição de mecanismos de teste (BERGER, 2002):

- Sistemas embarcados devem executar de maneira confiável por longos períodos de tempo;
- São utilizados com frequência em aplicações em que a vida humana está em risco;
- São muitas vezes tão sensíveis ao custo que não há margem para ineficiências;
- Devem com frequência compensar falhas no hardware embarcado; e
- Eventos no mundo real são normalmente assíncronos e não determinísticos, fazendo com que simulações sejam difíceis e não confiáveis.

1.2 Motivação

Este projeto de mestrado está inserido no contexto do INCT/SEC [1] (Instituto Nacional de Ciência e Tecnologia em Sistemas Embarcados Críticos), o qual possui como objetivo agregar grupos de pesquisa de diversas áreas estratégicas relacionados com sistemas embarcados críticos para o desenvolvimento sustentável do país, estimulando o desenvolvimento da pesquisa científica, tecnológica e a inovação, além da formação de recursos humanos e difusão da ciência.

Sistemas embarcados encontram-se cada vez mais presentes no cotidiano, integrando grande parte dos sistemas utilizados. Esses sistemas podem ser encontrados em aparelhos eletrônicos, eletrodomésticos, sistemas médicos, aéreos, automotivos, robóticos, dentre outros. O incorreto funcionamento destes sistemas pode acarretar prejuízos como a perda de grandes quantias monetárias e até mesmo pode oferecer risco a vidas humanas (WOLF, 2008; EBERT; SALECKER, 2009).

Observa-se que muitos aspectos presentes em programas concorrentes são semelhantes aos encontrados em sistemas robóticos móveis, os quais precisam interagir com o meio e com as partes que o compõem. Entre as principais semelhanças encontradas entre programas concorrentes e sistemas robóticos podem-se destacar a comunicação e sincronização de tarefas. Um sistema para robô móvel é composto por diversos dispositivos como sensores e atuadores que

[1] INCT/SEC- http://www.inct-sec.org/br/

são executados paralelamente. A comunicação e sincronização entre as tarefas desempenhadas por cada um dos dispositivos, muitas vezes, é não determinística, assim como em programas concorrentes, em que há diversos processos/*threads* executando de forma concorrente. Considerando esses aspectos, surgiu a motivação de investigar o teste de integração em sistemas embarcados críticos para robôs móveis, buscando explorar questões que vêm sendo tratadas no teste de programas concorrentes.

1.3 Objetivos

O principal objetivo deste projeto de mestrado é o estudo sobre a adaptação dos critérios de teste definidos para programas concorrentes para o contexto de robôs móveis. Este estudo deve considerar a aplicabilidade e adaptabilidade dos critérios de teste de programas concorrentes para o teste de integração de sistemas robóticos móveis. Devem ser consideraradas aplicações reais desenvolvidas pelo LRM[2] (Laboratório de Robótica Móvel), em que são demonstrados a aplicabilidade, adaptabilidade dos critérios de teste e as diferenças existentes entre aplicações robóticas e concorrentes. Outro objetivo deste trabalho é o estudo de ferramentas que auxiliem na automatização da aplicação dos critérios de teste e na execução dos casos de teste dos critérios.

1.4 Estrutura

Este trabalho está organizado em seis capítulos. Este capítulo introduziu a área de pesquisa, apresentando a motivação para o desenvolvimento deste trabalho. No Capítulo 2 são apresentados os principais conceitos relacionados às áreas de envolvidas neste estudo: teste de software, teste de programas concorrentes e robôs móveis. No Capítulo 3 são apresentados trabalhos relacionados a este projeto de mestrado. No Capítulo 4 é apresentado a adaptação dos critérios de teste de programas concorrentes para o teste de integração de robôs móveis. No Capítulo 5 é apresentado um estudo de caso realizado com uma aplicação robótica real. No Capítulo 6 são apresentadas as conclusões deste estudo e os desdobramentos para trabalhos futuros.

[2] LRM - www.lrm.icmc.usp.br/

CAPÍTULO

2

FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações Iniciais

A robótica móvel é a ciência que percebe e manipula o mundo físico por meio de dispositivos computacionais. Nesses sistemas são necessários mecanismos para perceber e atuar no ambiente. Exemplos destes mecanismos são: sensores laser, câmeras de vídeo, GPS (*global positioning system*) e unidades de medida inercial (THRUN; BURGARD; FOX, 2005). Um dos desafios da robótica é a integração de informações vindas de todos os sensores que compõe o sistema, de maneira que possam ser gerados comandos para o controle dos diferentes componentes, garantindo que a tarefa seja executada de modo correto e sem colocar em risco tanto o robô quanto aqueles que o cercam. É de suma importância que o robô preserve a sua integridade, bem como também dos elementos presentes em seu ambiente como seres humanos, objetos, utensílios, etc. A ação de um robô deve ocorrer de modo ativo sobre um determinado objeto alvo, se realmente for prevista a sua ação sobre este objeto. Estas questões tornam claro o quanto o robô é um sistema embarcado de natureza crítica (WOLF *et al.*, 2009). Observam-se que muitos aspectos presentes em programas concorrentes são semelhantes aos encontrados em sistemas robóticos móveis, os quais precisam interagir com o meio e com as partes que o compõem. Entre as principais semelhanças encontradas entre programas concorrentes e sistemas embarcados para robôs móveis pode-se destacar a comunicação e sincronização de tarefas. Um sistema para robô móvel é composto por diversos dispositivos como sensores e atuadores que são executados paralelamente. A comunicação e sincronização entre as tarefas desempenhadas por cada um dos dispositivos, muitas vezes, é não determinística, assim como em programas concorrentes, em que há diversos processos/*threads* executando de forma concorrente. Assim, este capítulo aborda as principais áreas envolvidas nesta pesquisa, o teste de software, sistemas robóticos móveis, teste para sistemas robóticos e teste de software para programas concorrentes.

O capítulo está organizado da seguinte forma: na Seção 2.2 são descritos os principais

conceitos de teste de software. São apresentadas as técnicas usadas durante essa atividade, assim como sua importância na garantia da qualidade de produtos de software e do processo de desenvolvimento. Na Seção 2.3 são apresentados conceitos sobre robôs móveis e sobre os principais ambientes de desenvolvimento para esses sistemas, com foco no ROS (*Robotic Operating System*), apresentando sua arquitetura e também os ambientes de teste disponíveis para utilização neste framework. Na Seção 2.4 são apresentados conceitos sobre programas concorrentes, critérios de teste para programas concorrentes e as ferramentas de teste para programas concorrentes. Ao final, na Seção 2.5, são apresentadas as considerações finais deste capítulo.

2.2 Teste de Software

O processo de desenvolvimento de software envolve uma série de atividades, e apesar das técnicas, métodos e ferramentas empregados, erros no produto ainda podem ocorrer. Atividades agregadas sob o nome de Garantia da Qualidade de Software são introduzidas ao longo do processo de desenvolvimento de software. Entre estas, as atividades de VV&T tem como objetivo minimizar a ocorrência de erros e riscos associados ao produto. Validação refere-se ao conjunto de atividades que garantem que a aplicação desenvolvida corresponda aos requisitos do software; Verificação refere-se ao conjunto de atividades que garantem que uma aplicação implementa corretamente uma função específica; e o Teste é um elemento crítico da garantia de qualidade de software e representa a revisão final da especificação, projeto e código (MYERS, 2004).

No contexto de teste de software, os seguintes conceitos estão presentes: a) defeito (*fault*) é um passo, processo ou definição de dados incorretos; b) engano (*mistake*) é uma ação humana que produz um ou mais defeitos; c) erro (*error*) é um estado inconsistente ou inesperado produzido por um defeito; e d) falha (*failure*) é um evento notável em que o sistema viola sua especificação, produzindo uma saída incorreta e inesperada (DELAMARO; MALDONADO; JINO, 2007). Um defeito pode ser identificado em qualquer nível do processo de desenvolvimento do software dessa forma a atividade de teste pode ser realizada durante todo o ciclo de desenvolvimento do software. Para que isso ocorra a atividade de teste é dividida em três fases com objetivos distintos: unidade, integração e sistema (MYERS, 2004).

O teste de unidade tem como foco as menores unidades de um programa, que podem ser funções, procedimentos, métodos ou classes. Nesse contexto, esperam-se que sejam identificados erros relacionados a algoritmos incorretos ou mal implementados, estruturas de dados incorretas ou simples erros de programação (BARBOSA *et al.*, 2000).

O teste de integração deve ser realizado após testadas as unidades individualmente, a ênfase está na construção da estrutura do sistema. À medida que as diversas partes do software são colocadas para trabalharem juntas, é preciso verificar se a integração entre elas funciona de

maneira adequada (DELAMARO; MALDONADO; MARTHUR, 2001; BARBOSA *et al.*, 2000). No teste de integração algumas estruturas podem ser utilizadas para auxiliar a sua realização, dentre estas podem-se citar o ***Driver*** e o ***Stub***. O *Driver* é um componente de software que exercita as interfaces do módulo que chama o módulo que está sendo testado, fornecendo assim a entrada e controlando a saída para um dado caso de teste. O *Stub* é um programa simples que possui as mesmas interfaces de um módulo chamado pelo módulo de teste. Um *Stub* substitui outros módulos ainda não integrados.

Utilizando uma estrutura de projeto não incremental pode-se utilizar para o teste de integração a estratégia ***Big bang***, na qual o sistema é exercitado quando já está integrado. Todos os módulos são testados de forma unitária e na sequência os módulos são todos integrados e testados. As desvantagens desse tipo de teste são a difícil localização de defeitos, o tratamento dos módulos críticos é realizado como os demais módulos e os conjuntos de teste são geralmente pequenos para revelar defeitos nas interfaces.

As principais estratégias incrementais são ***Top-down***, ***Bottom-up*** e ***Sandwich***. Na estratégia *Top-down* os componentes são adicionados ao sistema numa ordem hierárquica, começando com os componentes de nível mais elevado, ou seja, com maiores dependências. O teste tem início com o módulo do programa principal. Os módulos de níveis mais baixos chamados pelo módulo principal são substituídos por *Stubs* à medida que o teste avança. Os *Stubs* são aos poucos substituídos pelo código e testes são novamente realizados. As vantagens apresentadas são que os módulos são depurados separadamente e o teste das principais funções de controle é realizado logo no inicio, fazendo com que problemas sejam revelados mais cedo. Nessa estratégia duas abordagens podem ser utilizadas:

- **Em largura:** inicia-se com a integração dos módulos com dependências;
- **Em profundidade:** integra-se todos os módulos que implementam uma determinada funcionalidade da estrutura de software.

Na estratégia *Bottom-up* os componentes são adicionados por etapas seguindo a ordem de dependência, iniciando com componentes que possuem menores dependências (ordem de baixo para cima). Nesse tipo de estratégia são usados *Drivers* para substituir e simular alguns módulos durante o teste. Duas vantagens dessa estratégia são que módulos são depurados separadamente e o projeto de casos de teste é mais fácil pela ausência de *Stubs*.

A *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 dos módulos, dessa maneira, os módulos acima da linha são testados usando a estratégia *Top-down* e os módulos abaixo da linha são testados usando *Bottom-up*.

Segundo Delamaro (1997) são estabelecidos três tipos de falhas que caracterizam um erro de integração. Suponha-se o programa **P**, o caso de teste t e as unidades f e g, tal que f faz

chamadas a g. Define-se $S_I(g)$ como um conjunto de valores passados de f para o g e $S_O(g)$ como o conjunto de valores retornados de g para F. Ao executar-se **P** com o caso de teste t identifica-se uma falha causada por um erro de integração das unidades f e g em uma das seguintes situações:

- Tipo 1: se na chamada de g $S_I(g)$ contém algum valor incorreto que leva a uma saída incorreta (uma falha) antes do retorno de g;
- Tipo 2: se na chamada de g $S_I(g)$ contém algum valor incorreto que faz com que $S_O(g)$ assuma algum valor incorreto levando a uma saída incorreta (uma falha) após o retorno de g;
- Tipo 3: se na chamada de g, todos os valores $S_I(g)$ são esperados, porém valores incorretos de $S_O(g)$ são produzidos em g, levando a uma saída incorreta (uma falha) após o retorno de g.

A Figura 1 apresenta uma visão geral dos três tipos de falhas que caracterizam erros na realização do teste de integração.

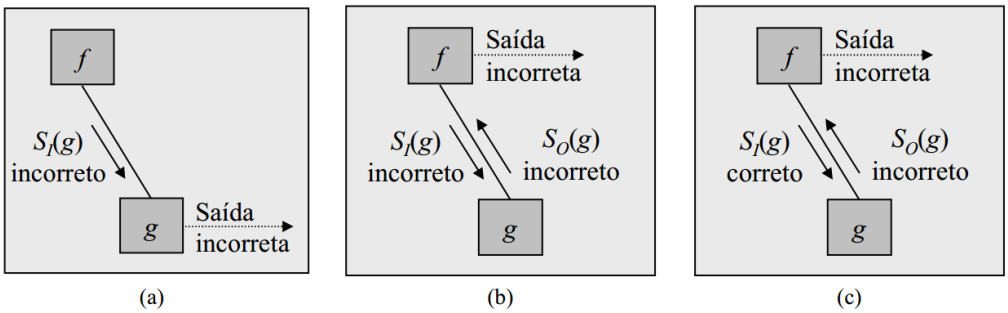

Figura 1 – Tipos de erros de integração: (a) Erro Tipo 1, (b) Erro Tipo 2 e (c) Erro Tipo 3 (DELAMARO, 1997).

O teste de sistema tem por objetivo verificar se as funcionalidades especificadas no documento de requisitos estão todas corretamente implementadas. Assim, aspectos como correção, completude e coerência devem ser explorados, bem como os requisitos não funcionais como segurança, performance e robustez (DELAMARO; MALDONADO; MARTHUR, 2001).

Além dessas três fases de teste, pode-se destacar também o teste de regressão. Este tipo de teste não é realizado durante o processo de desenvolvimento, mas sim durante a manutenção do software. Nesse caso, o objetivo é garantir que as alterações realizadas durante a manutenção não tenham introduzidos novos defeitos no software (DELAMARO; MALDONADO; MARTHUR, 2001).

Durante a atividade de teste é necessário saber se os testes efetuados são realmente de boa qualidade. Sabe-se que o teste exaustivo é impraticável, ou seja, testar utilizando todos os dados possíveis do domínio de entrada é, em geral, caro e demanda muito mais tempo do que o disponível. Deve-se salientar que não existe um procedimento de teste de propósito geral

que possa ser usado para provar a corretitude de um programa. Mesmo que não seja possível, por meio de testes, provar que um programa está correto, os testes, se conduzidos sistemática e criteriosamente, contribuem para aumentar a confiança de que o software desempenha as funções especificadas e evidenciam algumas características mínimas do ponto de vista da qualidade do produto. Um critério de teste define propriedades ou requisitos que precisam ser considerados pelos testes para avaliar a sua qualidade (ZHU; HALL; MAY, 1997; DELAMARO; MALDONADO; MARTHUR, 2001). Critérios para selecionar e avaliar conjuntos de casos de teste são fundamentais para o sucesso da atividade de teste. Tais critérios devem fornecer indicação de quais casos de teste devem ser utilizados de forma a aumentar as chances de revelar erros ou, quando erros não forem revelados, estabelecer um nível elevado de confiança na correção do programa. Os critérios de teste de software são estabelecidos, basicamente, a partir de três técnicas:

- **Técnica funcional**: é uma técnica utilizada para o projeto de casos de teste na qual o programa ou sistema é considerado uma caixa preta e, para testá-lo, são fornecidas entradas e avaliadas as saídas geradas para verificar se estão em conformidade com os objetivos especificados (FABBRI; VINCENZI; MALDONADO, 2007). Nessa técnica, 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 esteja correta e bem escrita para tornar possível 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 não são avaliados detalhes de implementação do sistema. Os principais critérios dessa técnica são: o particionamento em classes de equivalência, análise do valor limite e o grafo causa-efeito (MYERS, 2004; OSTRAND; BALCER, 1988).

- **Técnica estrutural**: ou caixa branca, estabelece os requisitos de teste com base em uma dada implementação, requerendo a execução de partes ou de componentes elementares do programa. Os caminhos lógicos do software são testados, utilizando-se casos de teste que põem à prova tanto conjuntos específicos de condições e/ou laços como pares de definições e usos de variáveis. Os critérios de teste estruturais são, em geral, classificados em : (i) critérios baseados na complexidade; (ii) critérios baseados no fluxo de controle; e (iii) critérios baseados no fluxo de dados (DELAMARO; MALDONADO; JINO, 2007; WEYUKER; WEISS; HAMLET, 1991).

- **Técnica baseada em defeitos**: esta técnica utiliza os defeitos mais comuns que podem ocorrer durante o processo de desenvolvimento de software para derivar os requisitos de teste (MYERS, 2004). O objetivo desse critério é apoiar a geração de um conjunto de teste capaz de indicar que os defeitos inseridos não estão presentes no programa em teste, aumentando a confiança sobre a corretude desse programa (DEMILLO, 1987). O teste

de Mutação ou Análise de Mutantes é um critério da técnica baseada em erros, em que o artefato (programa ou especificação) em teste é alterado com a utilização dos operadores de mutação, como se defeitos estivessem sendo inseridos no mesmo. Os artefatos gerados com as mudanças modeladas pelos operadores de mutação são chamados mutantes. O objetivo é produzir casos de teste capazes de evidenciar os defeitos inseridos nos programas e desse modo melhorar a qualidade do conjunto de teste, indicando que os defeitos inseridos pelo teste de mutação não estão presentes no programa em teste.

É importante destacar que as técnicas são complementares, assim, ao se explorar diferentes técnicas é possível testar diferentes aspectos do programa e com isso melhorar a qualidade da atividade de teste de software. Devido à importância da técnica estrutural para este trabalho de mestrado, a mesma é abordada com maiores detalhes.

2.2.1 Técnica Estrutural

A técnica estrutural baseia-se no conhecimento da estrutura interna da implementação. Em geral, a maioria dos critérios dessa técnica utiliza uma representação de programa conhecida como Grafo de Fluxo de Controle (GFC). Nesse grafo, o programa *P* é decomposto em um conjunto de blocos separados de comandos, em que a execução do primeiro comando de um bloco ocasiona a execução de todos os outros comandos desse bloco, na ordem estabelecida (BARBOSA *et al.*, 2007).

O GFC é representando por G= N, E, s, onde N representa o conjunto de nós do programa, E o conjunto de arestas e s o nó inicial. Assim, o GFC consiste em estabelecer uma correspondência entre nós e indicar possíveis fluxos de controle no programa. A partir do grafo de programa podem ser escolhidos os componentes que devem ser executados, caracterizando assim o teste estrutural (BARBOSA *et al.*, 2007).

Critérios Baseados em Fluxo de Controle derivam seus requisitos de teste com base nas informações de controle existentes no programa, como comandos ou desvios, para determinar quais estruturas devem ser testadas (PRESSMAN, 2000; BARBOSA *et al.*, 2007). Alguns critérios desse grupo são:

- **Todos-Nós:** requer que a execução do programa passe, ao menos uma vez, em cada nó do GFC, ou seja, que cada comando do programa seja executado pelo menos uma vez pelo conjunto de casos de teste;
- **Todos-Arcos:** requer que cada aresta do grafo, ou seja, cada desvio de fluxo de controle do programa, seja exercitado pelo menos uma vez pelo conjunto de casos de teste.
- **Todos-Caminhos:** requer que todos os caminhos possíveis do GFC sejam executados pelo menos uma vez pelo conjunto de casos de teste;

- **Todos-Caminhos-Completos:** requer que sejam executados todos os caminhos completos. Um caminho completo é aquele em que o primeiro nó é inicial S e o último nó é nó de saída do GFC;

- **Todos-Caminhos-Simples:** requer que sejam executados todos os caminhos simples. O caminho simples é aquele em que todos os nós que o constituem são distintos, exceto possivelmente o primeiro e o último;

- **Todos-Caminhos-Livre-de-Laços:** requer que todos os caminhos livres de laços sejam percorridos ao menos uma vez pelo conjunto de testes. Caminho livre de laço é um caminho simples em que todos os nós são distintos inclusive o primeiro e o último.

Critérios Baseados em Fluxo de Dados utilizam informações sobre o fluxo de dados do programa para derivar os requisitos de teste. Esses critérios exploram as interações que envolvem definições de variáveis e referência a tais definições para estabelecerem os requisitos de teste (PRESSMAN, 2000; BARBOSA *et al.*, 2007).

O Grafo **Def-Uso** (*Def-Use Graph*) foi definido por Rapps e Weyuker (1985), o qual consiste em uma extensão do GFC, no qual são adicionadas informações a respeito do fluxo de dados do programa, caracterizando associações entre pontos do programa em que é atribuído um valor a uma variável (chamado de definição da variável) e pontos em que esse valor é utilizado (chamado de referência ou uso de variável).

Uma variável é usada quando a referência a essa variável não a estiver definindo. Cada ocorrência de variável é classificada como sendo uma definição (*def*), um uso-computacional (*c- uso*), ou um uso predicativo (*p-uso*). O *c-uso* afeta diretamente uma computação sendo executada ou permite que o resultado de uma definição anterior possa ser observado, podendo afetar indiretamente o fluxo de controle do programa; um *p-uso* afeta o fluxo de controle do programa e pode afetar indiretamente computações executadas (RAPPS; WEYUKER, 1985). Entre os critérios dessa família destacam-se os seguintes:

- **Todas-Definições:** requer que cada definição de variável seja exercitada pelo menos uma vez, não importa se por um *c-uso* ou por um *p-uso*;

- **Todos-Usos:** requer que todas as associações entre uma definição de variável e seus subsequentes usos (*c-usos* e *p-usos*) sejam exercitadas pelos casos de teste, por pelo menos um caminho livre de definição, ou seja, um caminho em que a variável não é redefinida. Uma associação pode ser definida como o par formado pela definição de uma variável e seu subsequente uso;

- **Todos-Du-Caminhos:** requer que toda associação entre uma definição variável e seus subsequentes *p-uso* ou *c-uso* seja exercitada por todos os caminhos livres de laço que cubram essa associação.

Os Critérios Baseados na Complexidade utilizam informações sobre a complexidade do programa para derivar os requisitos de teste. O critério de McCabe pode ser evidenciado como o critério mais conhecido dessa categoria. Ele utiliza a complexidade ciclomática do grafo do programa para derivar os requisitos de teste. Esse critério requer que um conjunto de caminhos linearmente independentes do grafo de programa seja executado (PRESSMAN, 2000; DELAMARO; MALDONADO; JINO, 2007).

A maior parte dos critérios baseados em fluxo de dados, para requerer um determinado elemento (caminho, associação, etc.), exige a ocorrência explícita de um uso de variável e não garante, necessariamente, a inclusão dos critérios Todos-Arcos na presença de caminhos não executáveis, presentes na maioria dos programas (BARBOSA *et al.*, 2000). Assim foram introduzidos os critérios Potenciais-Usos (MALDONADO, 1991). Esses critérios são caracterizados independentemente da ocorrência explícita de uma referência a um uso de uma determinada definição. Se existir um caminho livre de definição até certo nó ou arco um potencial-uso é caracterizado, e eventualmente é requerido. Da mesma forma como os demais critérios baseados na análise de fluxo de dados, os critérios Potenciais-Usos podem utilizar o Grafo Def-Uso como base para o estabelecimento dos requisitos de teste.

2.3 Sistemas Robóticos Móveis

Segundo Wolf *et al.* (2009), os robôs móveis são umas das aplicações em maior ascensão nos últimos anos em sistemas embarcados. No passado o destaque era para robôs industriais e braços mecânicos robóticos, com o passar dos anos as expectativas mudaram um pouco e atualmente destaca-se o desenvolvimento de robôs móveis autônomos (RMA) e veículos autônomos inteligentes, os quais devem ser capazes de interagir com o ambiente, aprender e tomar decisões corretas para que suas tarefas sejam executadas com êxito. A robótica móvel é a ciência que percebe e manipula o mundo físico por meio de dispositivos computacionais. Nesses sistemas são necessários mecanismos para perceber e atuar no ambiente. Exemplos destes mecanismos são: sensores laser, câmeras de vídeo, GPS (*global positioning system*) e unidades de medida inercial (THRUN; BURGARD; FOX, 2005).

Bekey (2005) afirma que autonomia refere-se a sistemas capazes de operar em ambientes reais, sem qualquer forma de controle externo por longos períodos de tempo. Neste sentido, deve-se considerar que maiores níveis de autonomia serão alcançados somente à medida que o robô passe a integrar outros aspectos considerados de maior importância, como: capacidade de percepção (sensores que conseguem ler o ambiente onde ele atua), capacidade de agir (atuadores e motores capazes de produzir ações, tais como o deslocamento do robô no ambiente), robustez e inteligência (capacidade de lidar com as mais diversas situações, de modo a resolver e executar tarefas por mais complexas que sejam). O uso de técnicas de planejamento e controle robusto para a navegação e operação autônoma dos robôs é conhecido como Controle Robótico Inteligente.

O controle inteligente permite dotar os RMAs da capacidade de executar as mais diversas e complexas tarefas (WOLF *et al.*, 2009).

Robôs móveis são desenvolvidos visando substituir os seres humanos em atividades hostis que apresentam riscos à vida humana. Neste sentido, pesquisas são conduzidas para o desenvolvimento de robôs para patrulhamento interno, com aplicações em segurança, transporte, saúde, execução de tarefas em ambientes aquáticos, no espaço, exploração de cavernas e vulcões (PESSIN *et al.*, 2010; WOLF *et al.*, 2009). Outra aplicação direta dessa tecnologia é o desenvolvimento de sistemas de direção autônoma, os quais auxiliam na navegação de veículos em ruas e estradas (SHINZATO; OSORIO; WOLF, 2011; SOUZA *et al.*, 2011; FERNANDES *et al.*, 2012). A navegação é considerada um dos elementos fundamentais da robótica móvel, sendo a base para a execução de outras tarefas. Além dos desvios de obstáculos é necessário que o robô identifique o terreno navegável. Assim sendo, a combinação de dados obtidos por lasers, câmeras de vídeo e outros sensores devem ser adequadamente processadas por algoritmos, possibilitando contribuir para o reconhecimento da área navegável e propiciando autonomia ao robô.

Um dos desafios da robótica é a integração de informações vindas de todos os sensores que compõe o sistema, de maneira que possam ser gerados comandos para o controle dos diferentes componentes, garantindo que a tarefa seja executada de modo correto e sem colocar em risco tanto o robô quanto aqueles que o cercam. É de suma importância que o robô preserve a sua integridade, bem como também dos elementos presentes em seu ambiente como seres humanos, objetos, utensílios, etc. A ação de um robô deve ocorrer de modo ativo sobre um determinado objeto alvo, se realmente for prevista a sua ação sobre este objeto. Estas questões tornam claro o quanto o robô é um sistema embarcado de natureza crítica (WOLF *et al.*, 2009).

O INCT/SEC apoia o desenvolvimento de diversos projetos em sistemas embarcados, dentre estes está o CaRINA[1], apresentado na Figura 2. O projeto CaRINA (Carro Robótico Inteligente para Navegação Autônoma) tem como objetivo o desenvolvimento de um veículo autônomo inteligente capaz de navegar em ambientes urbanos sem a necessidade de um condutor humano. Outros objetivos do projeto são a redução do número de acidentes em ruas e rodovias e o desenvolvimento de um sistema de auxílio ao motorista, notificando-o de uma situação de risco durante a condução, bem como o controle do veículo quando o perigo é eminente. Outro projeto em desenvolvimento para robôs móveis é o VANT (Veículos Aéreos Não Tripulados). O objetivo deste projeto é o desenvolvimento e fabricação de veículos aéreos não tripulados (BRANCO *et al.*, 2011; TRINDADE *et al.*, 2010; WOLF *et al.*, 2009). Os VANTs são geralmente usados em aplicações críticas, pois permitem ser monitorados remotamente e são capazes de realizarem voos autônomos. Numa parceria entre a AGX[2] e o INCT/SEC foi desenvolvido o Tiriba, um pequeno VANT projetado para aplicações como agricultura, defesa ou medição da poluição urbana, dentre outras aplicações (BRAGA *et al.*, 2011).

[1] CaRINA - http://www.inct-sec.org/br/aplicacoes/carina
[2] AGX - http://www.agx.com.br

Figura 2 – Exemplo de robôs móveis: CaRINA veículo Palio Adventure.

Na sequência são apresentados os ambientes de desenvolvimento para robôs móveis, com maior ênfase no ROS (*Robotic operating System*), com um exemplo de sua utilização.

2.3.1 Metodologias para Desenvolvimento de Robôs Móveis

Para o desenvolvimento de robôs móveis autônomos são utilizados ambientes que permitem simular o comportamento dos robôs à medida vão sendo desenvolvidos. Alguns desses ambientes de desenvolvimento exigem uma licença para sua utilização, podendo ser inclusive comercializados juntamente com alguns modelos de robôs comerciais, enquanto outros podem ser utilizados livremente (WOLF *et al.*, 2009). A seguir são descritos alguns ambientes de desenvolvimento para robôs móveis.

- **Player**[3] é um sistema para controle de robôs móveis de código aberto e de livre distribuição e é utilizado por universidades e institutos de pesquisa (COLLETT; MACDONALD, 2005). A estrutura do Player é baseada no modelo cliente/servidor. Nesse modelo o servidor faz a interface com o robô e com outros sensores, obtendo dados, enviando-os para o cliente e recebendo instruções do cliente para o controle do robô e dos sensores. O cliente é o programa que controla efetivamente o robô (aplicação/sistema de controle) e é responsável por obter os dados do servidor, interpretá-los e enviar instruções de volta para o servidor (robô) durante a execução de uma determinada tarefa. A arquitetura cliente/servidor permite grande versatilidade no controle de robôs e sensores, de forma que um cliente (programa de controle) pode controlar diversos servidores e diferentes clientes podem controlar

[3] Player - http://playerstage.sourceforge.net/

diferentes sensores de um mesmo robô. O cliente Player foi projetado para ser compatível com diversas linguagens, como C, C++, Java, Python, entre outras. Além de fornecer funções para o controle de robôs e sensores, o Player também disponibiliza algoritmos para navegação com desvio de obstáculos, planejamento de trajetória, localização e mapeamento de ambientes (GERKEY; VAUGHAN, 2000; GERKEY; VAUGHAN; HOWARD, 2003; COLLETT; MACDONALD, 2005; XAVIER *et al.*, 2005). O Player possui dois outros módulos de simulação para robôs móveis: o Stage e o Gazebo (WOLF *et al.*, 2009).

O Stage [4] é um simulador de múltiplos robôs, em que vários robôs e sensores podem ser simulados simultaneamente e controlados por um ou mais clientes. O Stage simula o deslocamento dos robôs e também o comportamento de sensores, como odômetros, lasers, sonares, câmeras e garras.

O Gazebo [5] é um simulador de robôs móveis em 3D que permite uma simulação e visualização mais realista de ambientes complexos. Por meio da utilização de bibliotecas gráficas e de modelagem física, o Gazebo permite uma simulação fiel do comportamento e da interação física de robôs e objetos do ambiente. Quanto aos sensores, além daqueles simulados pelo Stage, o Gazebo simula também GPS, unidades de medida inercial, câmeras estéreo, entre outros.

- **CARMEN** [6] (*Carnegie Mellon Robot Navigation Toolkit*) é uma biblioteca modular e de código aberto para o controle de plataformas robóticas e sensores comerciais. Da mesma forma que o Player, o CARMEN disponibiliza funções de acesso ao hardware de robôs e sensores, bem como algoritmos normalmente utilizados em aplicações robóticas como navegação, mapeamento e localização.

- **ARIA** [7] (*Advanced Robotics Interface for Applications*) é um sistema desenvolvido para o controle dos robôs fabricados pela empresa MobileRobots. Dentre os robôs fabricados pela MobileRobots, destaca-se o Pioneer, que é amplamente utilizado para testes experimentais em diversas universidades e institutos de pesquisa. O MobileSim é um simulador de ambientes bidimensionais para robôs móveis compatível com o controlador ARIA. Esse simulador é baseado no código do Stage, com algumas modificações feitas pela empresa MobileRobots, dentre elas o suporte para ambientes Windows.

- ***Microsoft Robotics Studio (MRS)*** é uma ferramenta distribuída sob licenças *professional* ou *academic express*, podendo ser obtidas gratuitamente, porém esta não é uma ferramenta de código aberto. O controlador/simulador de robôs desenvolvidos pela Microsoft é compatível com o ambiente Windows e com suas ferramentas visuais de programação .NET.

[4] Stage http://playerstage.sourceforge.net/index.php?src=stage
[5] Gazebo http://gazebosim.org/
[6] CARMEN - http://carmen.sourceforge.net/
[7] ARIA - http://robots.mobilerobots.com/wiki/ARIA

- **Webots** [8] é um controlador/simulador, desenvolvido pela empresa Cyberbotics em conjunto com pesquisadores do *Swiss Federal Institute of Technology*. O sistema vem sendo utilizado em mais de 450 universidades e institutos de pesquisa e é compatível com plataformas Linux, Windows e Macintosh e linguagens C, C++ e Java (WOLF *et al.*, 2009).

- **OPRoS** [9]: (*The Open platform for Robotic Services*) é uma ambiente de desenvolvimento para robôs móveis, baseado em componentes. O OPRoS possui integração com diversas IDEs (*Integration Development Environment*) de desenvolvimento. O OPRoS também possui um framework para execução dos robôs, um servidor e possui integração com ferramentas de teste e verificação.

- **ROS:** é um *framework* popular na comunidade de robótica, baseado em componentes e que oferece comunicação e outros mecanismos necessários a implementação de software para robôs. O ROS é um ambiente de desenvolvimento que oferece bibliotecas e um conjunto de ferramentas para auxiliar os desenvolvedores de aplicações robóticas, possibilitando abstrair o hardware e drivers de dispositivo, de modo a simular o funcionamento da aplicação. O ROS é um dos ambientes adotados para o desenvolvimento dos projetos CaRINA I e CaRINA II (FERNANDES *et al.*, 2012).

Devido a sua importância neste trabalho, na próxima seção é apresentado com maiores detalhes o framework de desenvolvimento ROS [10].

2.3.2 Robotic Operating System - ROS

Dentre as principais características do ROS, podem ser citadas (QUIGLEY *et al.*, 2009):

- ***Peer-to-peer***: um sistema desenvolvido usando o *ROS* consiste de uma série de processos, em um número de *hosts* diferentes, conectados em tempo de execução caracterizando uma topologia *Peer-to-peer*;

- **Multi-linguagem**: o ROS utiliza uma interface de definição da linguagem, possibilitando que desenvolvedores utilizem diversas linguagens para o desenvolvimento. Algumas das linguagens utilizadas são C++, Phyton, Lisp e Octave;

- **Baseado em ferramentas**: para auxiliar no gerenciamento de complexidade do ROS é utilizado um *microkernel*, onde um grande número de ferramentas são usadas para construir e executar vários componentes. Essas ferramentas efetuam várias tarefas como, por exemplo, obter e definir os parâmetros de configuração, visualizar a topologia de

[8] Webots - http://www.cyberbotics.com/overview
[9] OPRoS - http://www.opros.or.kr/
[10] ROS - http://ros.org/wiki/

conexão da rede *Peer-to-peer*, demonstrar graficamente os caminhos das mensagens e diversos outros recursos;

- **Leve**: o desenvolvimento dos drivers e algoritmos ocorrem em bibliotecas de forma independente. Toda complexidade é colocada nas bibliotecas e são criados pequenos executáveis que apresentam as funcionalidades do ROS, facilitando a extração de código e a sua reutilização.

A Figura 3 ilustra uma visão do sistema de arquivos do ROS. No ROS as packages são organizadas em stacks. O objetivo de packages é criar coleções mínimas de código para facilitar a reutilização, a meta de stacks é simplificar o processo de compartilhamento de código. Stacks são o principal mecanismo do ROS para a distribuição de software. Cada stack tem uma versão associada e pode declarar dependências em outros stacks. Os conceitos fundamentais da implementação do sistema de arquivos do ROS são (QUIGLEY *et al.*, 2009; FERNANDES, 2012):

- **Nós**: são processos executáveis que utilizam o ROS para comunicação com outros nós, sendo que um sistema típico do ROS pode ser composto por muitos nós.
- **Mensagens**: são tipos de dados em assinaturas e publicações, assim os nós se comunicam pelo envio e recebimento de mensagens.
- **Tópico**: é o mecanismo de comunicação entre nós que utilizam o paradigma *publish/subscribe*. Um nó publica mensagens em um tópico. Podem haver vários nós publicando ou sobrescrevendo um tópico. Um ou mais nós podem sobrescrever mensagens publicadas em um tópico. No geral, os nós assinantes não estão cientes da existência dos nós que publicaram os dados.

Para instalação e utilização do ROS, o site oficial [11] disponibiliza o procedimento completo. Após a correta instalação de uma de suas versões é necessário a configuração de suas variáveis de ambiente. O ROS possui algumas funções com o objetivo de facilitar o gerenciamento e utilização de pacotes. Estas funções podem ser utilizadas pela linha de comando, dentre as principais pode-se citar: ***roscd*** que permite navegar diretamente para diretórios em que se encontram os pacotes; ***rosmake*** que compila automaticamente o pacote especificado; ***rostopic*** permite publicar informações em um tópico e mensagem específica; ***rxgraph*** que permite de maneira gráfica visualizar os pacotes em execução no ROS, tópicos e informações das mensagens.

Na Figura 4 é apresentado um exemplo de uma aplicação no ROS, utilizando a ferramenta ***rxgraph*** . Este exemplo foi criado no ROS, com objetivo verificar como é realizada a troca de mensagens entre os nós do sistema. Neste exemplo os nós são: *central_control*, *lighthouse*, *obstacle_detection*, *sensor*, *gps*, *march*, *break* e *throttle*. Os tópicos são: *exchange_march*,

[11] ROS - http://wiki.ros.org/

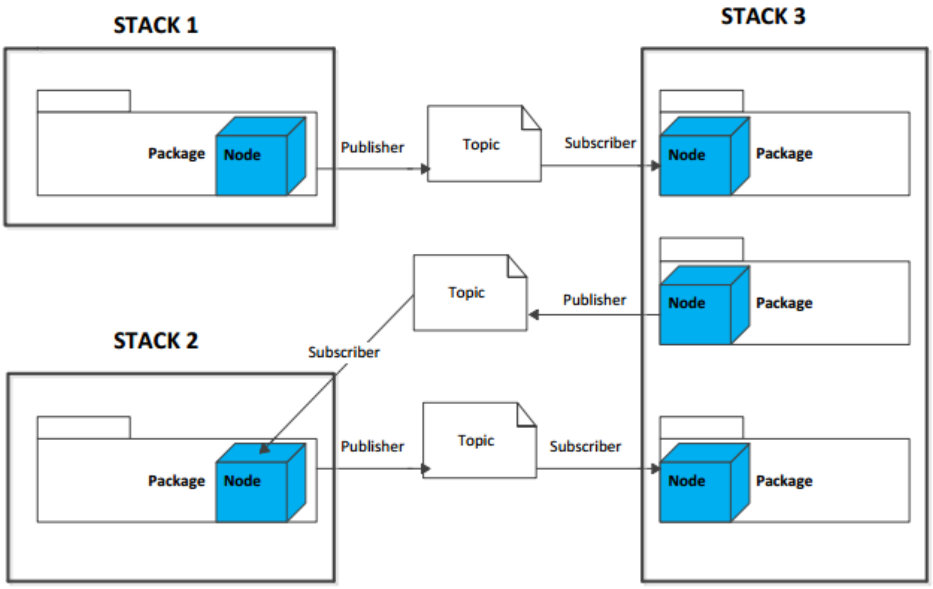

Figura 3 – Organização do sistema de arquivo do ROS (FERNANDES, 2012).

light_on,*light_off*, *obstacles*, *steering_commands*, *vehicle_state*, *velocity_commands*, *breake_-commands* e *throttle_commands*. Os nós podem enviar e receber mensagens utilizando os tópicos. Um exemplo dessa comunicação é apresentado na Figura 4, em que o nó *obstacle_detection* pode utilizar-se do tópico *obstacles* para publicar mensagens, assim, o nó *central_control* poderá ler estas mensagens. Outro exemplo é o nó *central_control*, que pode utilizar do tópico *brake_-commands* para publicar mensagens, assim, estas mensagens podem ser lidas pelo nó *break*.

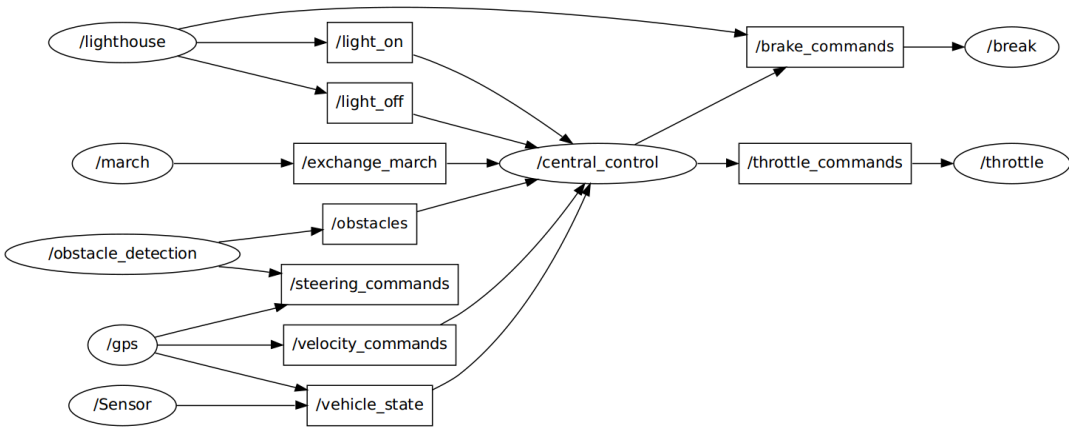

Figura 4 – Grafo de visualização de um sistema ROS gerado pela ferramenta *rxgraph*.

2.3.2.1 Paradigma Publisher/Subscriber

Para a comunicação entre componentes o ROS pode utilizar o paradigma *publish/Subscribe*. Um *publish* no Ros é assíncrono e funciona caso haja *subcribes* assinando ao tópico que está sendo publicado. Um *publish* é desenvolvido para fazer seu trabalho de maneira rápida e simples. O trabalho de um *publish* é:

- Serializar a mensagem para um *buffer*; e
- Enviar a mensagem para uma fila para posterior processamento.

Em um *publish/subscribe* caso a fila das mensagens que são enviadas ao tópico esteja cheia, a mensagem mais antiga é descartada antes de adicionar a nova mensagem. O Código-fonte 1 exemplo de publicação de dados utilizando o tópico *topic_name* e tipo de mensagem $< std_msg :: String >$ para uma fila de tamanho 5. Para que um dado seja publicado em um tópico, é necessário definir um identificador (*handle*) e usar o método "NodeHandle::advertise()"que criará um "ros::Publisher", declarando o nome do tópico que a mensagem será publicada e o tipo de dado. O identificador "ros::Publisher"chama o método *.publish()* passando a mensagem a ser publicada.

Código-fonte 1: Exemplo de código de um publisher()

```
1  ros::Publisher pub = nh.advertise<std_msgs::String>("
    topic_name", 5);
2   std_msgs::String str;
3   str.data = "hello world";
4   pub.publish(str);
```

Quando a mensagem é enviada, o nó *subscribe* para esse tópico definiu um ros::Subscriber no qual especificou de qual tópico ele subscreve mensagens e o nome do método a ser ativado quando um dado for publicado nesse tópico. Essa informação é usada pelo nó Mestre para saber quem subscreve de quem. Assim, quando a mensagem é enviada o método *callback* daquele tópico é ativado e recebe o dado. Um exemplo de um *subscribe* é apresentado no Código-fonte 2:

Código-fonte 2: Exemplo de código de um subscriber()

```
1     void callback(const std_msgs::StringConstPtr& str)
2     {
3     ...
4     }
5
6     ...
7     ros::Subscriber sub = nh.subscribe("my_topic", 1, callback)
    ;
```

A definição de um *subscribe* possui algumas informações similares às utilizadas em um *publish*. Em relação às diferenças, pode-se citar a utilização de uma função *callback* para monitoramento do recebimento de mensagens no tópico. A função *callback* é utilizada para monitorar o recebimento das mensagens que são publicadas no tópico e determina o consumo para o subscribe.

2.3.2.2 Ambientes de Testes no ROS

O ROS permite o desenvolvimento em diversas linguagens. Para algumas em específico, como Python e C++, é possível a integração com outros frameworks de teste para a condução de testes de unidade e de integração. O ROS disponibiliza uma ferramenta para gerenciamento dos testes, chamada **Rostest**. O Rostest é uma extensão para o *roslaunch*, o qual permite utilizar as características do *roslaunch* para realização de testes. O *roslaunch* é uma ferramenta que possibilita entre diversas confirgurações, a definição para execução de múltiplos nós. Esta ferramenta possui diversas informações, nas quais são configurados os parâmetros dos nós e locais em que eles serão executados. O Rostest possibilita a configuração e teste de um ou vários nós em execução, permite também que sejam configurados alguns testes para os nós, como tempo máximo que o nó deve executar antes que o teste seja encerrado como falho. Em conjunto ao Rostest podem ser utilizados o Gtest (*Google's framework for writing C++ tests*) e UnitTest (*Unit testing framework*) .

Para escrita dos testes com Rostest, é utilizada a tag *<test>*. Esta tag é sintaticamente similar a tag *<node>* e especifica o nó do ROS que será executado com o teste, além de indicar que o nó é um nó de teste que deve ser executado. Um exemplo de configuração básica de um *launch* para teste é apresentado na Código-fonte 3. Na figura a linha 1 possui a tag *<launch>* que indica o início do *launch*. Da linha 2 até linha 5 são apresentas as configurações do Nó que será testado. Da linha 6 até 10 são apresentadas as configurações dos testes, em que possui as informações de qual nó será testado, qual pacote o mesmo se encontra e o tempo limite para realização daquele teste.

Código-fonte 3: Exemplo de configuração de um ROSTEST.

```
1  <launch>
2   <node name="proximity"
3          pkg="rostest_node_example"
4          type="proximity"
5          required="true" output="screen"/>
6  <test test-name="TestProximity"
7          pkg="rostest_node_example"
8          type="test_proximity"
9          time-limit="60.0"/>
10 </launch>
```

O Gtest [12]é um framework do google para escrever testes em C++. Este framework é baseado na arquitetura xUnit. Utilizando o GTest é possível realizar testes automáticos, onde possui disponível um grande número de asserções que podem ser configuradas para utilização nos testes. No Código-fonte 4 é apresentado um exemplo de teste criado utilizando o Gtest, onde é demonstrado a configuração básica de um caso de teste para um Nó. Utilizando o Gtest com Rostest é possível a criação de testes unitários e de integração no ROS.

Código-fonte 4: Exemplo de configuração de código Gtest.

```
1 #include "gtest/gtest.h"
2 #include "ros/ros.h"
3 #include "std_msgs/Int16.h"
4 #include "std_msgs/Int8.h"
5
6 TEST_F(Callback, Test_Node_Proximity) {
7
8     loop_rate.sleep();
9     ros::spinOnce();
10
11    EXPECT_EQ(5, this->get_Proximity());
12    EXPECT_EQ(4, this->get_Proximity());
13 }
```

O UnitTest [13] é um framework de teste da família xUnit, criado especificamente para realização de testes unitários em códigos de linguagem Python e sua origem foi inspirada pelo JUnit. O UnitTest quando utilizado junto ao Rostest permite a criação e execução de testes unitários e de integração no ROS.

No geral, utilizando os frameworks Rostest + Gtest ou UnitTest, podem ser realizados testes nos seguintes níveis no ROS.

- Testes unitários: são realizados nas menores unidades do sistema, como métodos, funções, classes e etc. Para realização destes testes podem ser utilizados os frameworks Gtest e UnitTest e não necessitam da utilização do Rostest.

- Testes de nós: no ROS os nós podem ser testados individualmente, para simular o seu comportamento e verificar se internamente estão agindo como esperado para publicação e sobrescrever dados de tópicos. Para realização do teste neste nível, é necessária a utilização do ROSTEST em conjunto com Gtest ou UnitTest. Este nível de teste permite realizar também testes em API externas como tópicos assinados, tópicos publicados e serviços.

[12] https://code.google.com/p/googletest/
[13] UnitTest: http://pythontesting.net/books/python-testing-ebook/

- Teste de integração: neste nível de teste também é necessária a utilização do Rostest em conjunto com Gtest ou UnitTest. Os Nós são testados em conjunto, simulando a execução do sistema. Para testar a integração dos nós, vários nós são iniciados e é verificado se estes nós trabalham junto como esperado. Como um sistema robótico é constituído de diversos nós, debugar uma coleção de nós no ROS é como debugar um sistema *multithread*, pois pode haver *deadlock*, condições de disputa, etc.

Um exemplo de cada um destes níveis é apresentado na Figura 5. Neste cenário há dois nós que precisam ser testados. Cada nó possui seu próprio código. No primeiro nível de teste seria usado o Gtest ou UnitTest para realização do teste de unidade, representado na parte (A). Neste nível o teste é realizado diretamente no código. O segundo nível de teste é realizado nos componentes, em que é testado o componente, porém não interessa a estrutura interna do componente e sim suas interfaces, verificando assim casos extremos, erros de entradas ou saídas, etc. O segundo nível é apresentado na parte (B) e para realização do teste é utilizado RosTest + Gtest/UnitTest. O terceiro nível é o de integração, onde os múltiplos nós são testados trabalhando junto. Este teste tem por objetivo verificar como os nós se comportam quando em execução com os demais. Nesta fase é utilizado o RosTest + Gtest/UnitTest. Este projeto esta inserido na realização de testes nos níveis B e C.

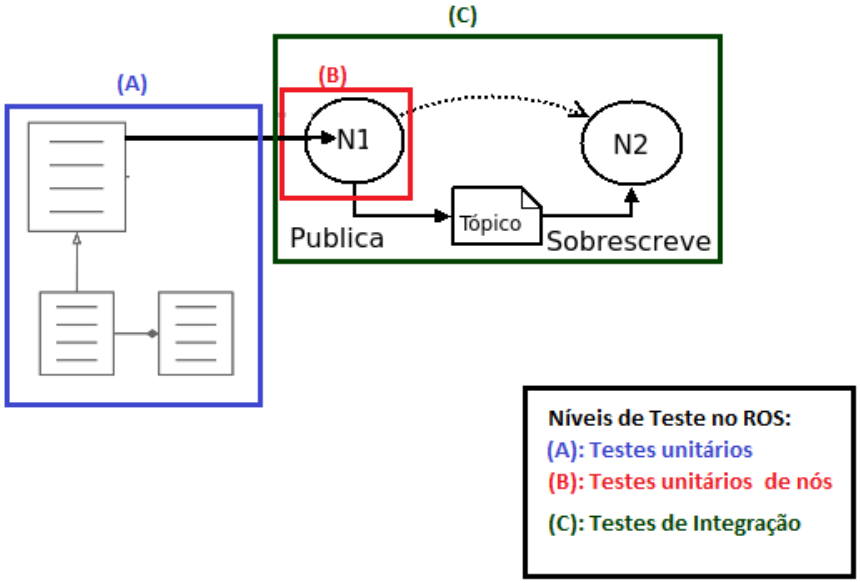

Figura 5 – Níveis de teste no ROS

2.4 Teste Estrutural de Programas Concorrentes

Na Seção 2.2.1 foi apresentado sobre teste estrutural para programas sequenciais. Nesta seção são apresentados os critérios propostos para programas concorrentes. Estes critérios

serão estudados e mapeados para aplicações robóticas. O teste de programas concorrentes deve considerar as particularidades dessas aplicações, incluindo comportamento não determinístico, comunicação e sincronização.

O grupo de Engenharia de Software e Sistemas Distribuídos/Programação Concorrente do ICMC/USP tem pesquisado o mapeamento de critérios de teste sequenciais para este contexto, considerando informações sobre o fluxo de controle, dados, comunicação e sincronização desses programas. O início das pesquisas ocorreu com a definição de critérios estruturais para programas concorrentes com memória distribuída (SOUZA *et al.*, 2005). Um modelo de teste baseado no fluxo de controle e no fluxo de comunicação de programas paralelos é estabelecido para extrair informações relevantes para a realização dos testes. Este modelo foi baseado no trabalho de Yang, Souter e Pollock (1998) e é composto por um grafo denominado PGFC (*Parallel Control Flow Graph*) para o contexto de programas com passagem de mensagem.

Dado um programa *Prog*, sendo este programa paralelo e constituído pelo conjunto de processos $Prog = p^0, p^1, ...p^{n-1}$. Cada processo p possui seu GFC, construído utilizando-se os mesmos conceitos utilizados para programas sequenciais. Um nó n pode ou não estar associado a uma função de comunicação (*send*() ou *receive*()). Considera-se que as funções de comunicação possuem os mesmos parâmetros da seguinte forma: *send*(i,j,t) (ou *receive* (i,j,t)), significando que o processo i envia ou recebe a mensagem t para o (ou do) processo j. Cada GFC possui um único nó de entrada e um único nó de saída que representam o início e o término da execução do processo p (SOUZA; VERGILIO; SOUZA, 2007).

Um Grafo de Fluxo de Controle Paralelo (GFCP) é construído para um programa concorrente *Prog*, sendo composto pelos GFC^p (para p=0...n-1) e pelas arestas de comunicação dos processos paralelos. N e E representam, respectivamente, o conjunto de nós e arestas de GFCP. Um nó i em um processo p é representando pela notação: n_i^p. Dois subconjuntos de N são definidos: N_s e N_r, compostos por nós que possuem, respectivamente, funções de envio e funções de recebimento de mensagens. Para cada $n_i^p \in N_s$, um conjunto R_i^p é associado, o qual contém os possíveis nós que recebem mensagens enviadas pelo nó n_i^p.

As arestas do GFCP podem ser de dois tipos: arestas intraprocessos (E^i), que são internas a um processo P e arestas interprocessos (E^s), que representam a comunicação entre os processos, ou seja, são arestas que ocorrem entre processos paralelos (SOUZA; VERGILIO; SOUZA, 2007). Um caminho π no GFC de um processo P é chamado intraprocesso se ele não contém nenhuma aresta interprocessos. É dado por uma sequência de nós, $\pi = (\mathrm{n}_1, n_2, ..., n_m)$ tal que $(n_i, n_i + 1) \in E_i^p$. Um caminho π no GFCP que contém pelo menos uma aresta interprocesso é dito um caminho interprocessos. Um caminho interprocessos é completo se todos os caminhos intraprocessos que o compõem são também completos (SOUZA; VERGILIO; SOUZA, 2007).

Em programas escritos no ambiente de passagem de mensagem, um novo tipo de definição de variável é possível: a definição decorrente da execução de uma função de comunicação tal como uma primitiva *receive*. O conjunto de variáveis definidas em um nó n_i^p é dado por $def(\ n_i^p)$.

A utilização de uma variável x ocorre quando x é referenciada em uma expressão. No contexto de programas concorrentes podem ocorrer três formas de uso de variáveis:

- **c-uso:** *uso computacional*, quando há um uso de uma variável em um nó n_i^p do grafo;
- **p-uso:** *uso predicativo*, quando há um uso de uma variável associada a um predicado em arestas intraprocessos (n_i^p, n_j^p) do grafo;
- **s-uso:** *uso comunicacional*, quando há um uso de uma variável em comandos de sincronização (*send* e *receive*, em arestas interprocessos $(n_i^{p1}, n_j^{p2}) \in E_s$.

Partindo dessas informações, é possível caracterizar as associações que devem ser requeridas pelos critérios estruturais aplicados a programas concorrentes. Esse modelo possui as seguintes associações:

- **associação c-uso:** que é dada pela tripla (n_i^p, n_j^p, x) em que $x \in def(n_i^p)$, e n_j^p contém um *c-uso* de x e há um caminho livre de definição com relação a x de n_i^p para n_j^p;
- **associação p-uso:** que é dada pela tripla $(n_i^p, (n_j^p, n_k^p)\ x)$ em que $x \in def(n_i^p)$, e (n_j^p, n_k^p) possui um p-uso de x e existe um caminho livre de definição com relação a x de n_i^p para (n_j^p, n_k^p);
- **associação s-uso:** que é dada pela tripla $(n_i^p, (n_j^{p1}, n_k^{p2})\ x)$ em que $x \in def(n_i^{p1})$, e (n_j^{p1}, n_k^{p2}) possui um s-uso de x e existe um caminho livre de definição com relação a x de n_i^{p1} para (n_j^{p1}, n_k^{p2}).

As associações p-uso e c-uso são associações intraprocessos, isto é, a definição e o uso de x ocorrem num mesmo processo p_1; uma associação s-uso requer uma comunicação entre processos, caracterizando assim uma associação interprocessos. Estas permitem identificar erros de envio e recebimento de mensagens. Considerando associações s-usos e a comunicação entre processos, outros tipos de associações entre processos são caracterizadas:

- associação s-c-uso: que é dada pela quíntupla $(n_i^{p1}, (n_j^{p1}, n_k^{p2}), n_l^{p2}, x^{p1}, x^{p2})$, tal que existe uma associação s-uso $(n_i^{p1}, (n_j^{p1}, n_k^{p2})\ x^{p1})$ e uma associação c-uso $(n_k^{p2}, n_l^{p2}, x^{p2})$;
- associação s-p-uso: que é dada pela quíntupla $(n_i^{p1}, (n_j^{p1}, n_k^{p2}), (n_l^{p2}, n_m^{p2}\ )\ x^{p1}, x^{p2})$, tal que existe uma associação s-uso $(n_i^{p1}, (n_j^{p1}, n_k^{p2}), x^{p1})$ e uma associação p-uso $(n_k^{p2}, (n_l^{p2}, n_m^{p2})$ $x^p 2)$.

A partir desse modelo de definições Vergilio, Souza e Souza (2005) definem duas famílias de critérios de testes estruturais para programas concorrentes em ambientes de passagem de mensagem:

1. **Critérios baseados no fluxo de controle e no fluxo de comunicação:**

   Cada GFC pode ser testado individualmente utilizando-se os critérios Todos-Nós e Todas-Arestas. Entretanto, como o objetivo é testar as comunicações presentes no GFCP para um dado *Prog*, os seguintes critérios são definidos:

   - **Todos-Nós-s:** requer que todos os nós do conjunto N_s sejam executados pelo menos uma vez pelo conjunto de casos de teste;
   - **Todos-Nós-r:** requer que todos os nós do conjunto N_r sejam exercitados pelo menos uma vez pelo conjunto de casos de teste;
   - **Todos-Nós:** requer que todos os nós do conjunto N sejam exercitados pelo menos uma vez pelo conjunto de casos de teste;
   - **Todas-Arestas-s:** requer que todas as arestas do conjunto E_s sejam exercitados pelo menos uma vez pelo conjunto de casos de teste;
   - **Todas-Arestas:** requer que todas as arestas do conjunto E sejam exercitadas pelo menos uma vez pelo conjunto de casos de teste.

2. **Critérios baseado no fluxo de dados e em passagem de mensagens:**

   - **Todas-Defs:** requer que para cada nó n_i^p e cada x em $def(n_i^p)$ uma associação definição-uso (c-uso ou p-uso) com relação a x seja exercitada pelo conjunto de casos de teste;
   - **Todas-Defs/s:** requer que para cada nó n_i^p e cada x em $def(n_i^p)$ uma associação s-c-uso ou s-p-uso com relação a x seja exercitada pelo conjunto de casos de teste;
   - **Todos-c-Usos:** requer que todas as associações c-usos sejam exercitadas pelo conjunto de casos de teste;
   - **Todos-p-Usos:** requer que todas as associações p-usos sejam exercitadas pelo conjunto de casos de teste;
   - **Todos-s-Usos:** requer que todas as associações s-usos sejam exercitadas pelo conjunto de casos de teste;
   - **Todos-s-c-Usos:** requer que todas as associações s-c-usos sejam exercitadas pelo conjunto de casos de teste;
   - **Todos-s-p-Usos:** requer que todas as associações s-p-usos sejam exercitadas pelo conjunto de casos de teste.

Um novo modelo chamado PCFGsm (*Parallel Control Flow Graph for Shared Memory*) baseado no modelo PCFG anterior foi estabelecido por (SARMANHO, 2009), com critérios de teste estruturais para programas que utilizam o paradigma de memória compartilhada. Esse novo modelo suporta a análise estática do programa, a geração de elementos requeridos utilizando os

critérios de teste propostos em (SOUZA *et al.*, 2008), adaptados para o contexto de memória compartilhada e um método que registra a comunicação entre as *threads* por meio de sincronizações realizadas. Foi necessário definir um novo modelo para o contexto de memória compartilhada, pois o PCFG não se adaptava às características desses programas. Nesses programas há a necessidade de diferenciar a comunicação e sincronização entre os processos (ou *threads*). Além disso, as variáveis compartilhadas fazem parte do mesmo espaço de endereçamento, entretanto, devido ao não determinismo, não é possível determinar antes da execução os pares entre a definição e o uso das variáveis. Desse modo, informações sobre o fluxo de dados do programa não podem ser obtidas estaticamente da forma que é feita no modelo anterior.

Além de c-uso e p-uso de variáveis, os seguintes tipos são definidos por Sarmanho *et al.* (2008):

- ***sync-use*** (uso de sincronização): ocorre quando a variável x está associada a instruções de sincronização, a variável x é do tipo semáforo $x \in V_S$;
- ***comm-c-use*** (c-uso- comunicacional): ocorre quando há um uso computacional de uma variável compartilhada;
- ***comm-p-use*** (p-uso- comunicacional): ocorre quando há um uso predicativo de uma variável compartilhada.

2.5 Considerações Finais

Neste capítulo foi apresentada uma introdução ao teste e seus conceitos básicos. Foi contextualizada a área de sistemas robóticos móveis, em que foram apresentados os frameworks de desenvolvimento de aplicações robóticas, em que a ênfase foi o ROS. Foi abordado também o teste de programas concorrentes, apresentado os critérios de teste definidos para este tipo de programa e as ferramentas para teste de programas concorrentes.

O desenvolvimento deste capítulo teve como objetivo contextualizar a área de teste de software, teste de software para programas concorrentes e a área de sistemas robóticos móveis. Estes conceitos são importantes para a compreensão dos capítulos subsequentes.

CAPÍTULO

3

TRABALHOS RELACIONADOS

3.1 Considerações Iniciais

O teste de aplicações robóticas e embarcadas pode ser influenciado por diversos fatores, os quais podem dificultar a sua condução. Alguns problemas e dificuldades detectados para o teste destes sistemas são (LEE; KIM; CHA, 2002; LIER F., 2013):

- Nem todos os componentes são desenvolvidos utilizando a mesma linguagem de programação ou o mesmo sistema operacional;
- Aplicações robóticas estão expostas a interações do mundo real, fazendo com que simulações sejam difíceis e não confiável.
- Estes sistemas estão sujeitos a frequentes mudanças para atender às diversas demandas dos clientes;
- O tempo disponível para condução dos testes é geralmente limitado, devido em muitos casos, o curto tempo de vida dos produtos. Fabricantes são pressionados a estar sempre atualizando seus produtos no mercado, com novas e sofisticadas características. Embora a maior parte do sistema possa ser reutilizado de um modelo para outro, o teste exige mais atenção quando há reutilização de código fonte;
- Sistemas embarcados e robóticos, geralmente, consistem de um grande número de componentes em execução que interagem entre si. Cada um desses componentes é executado em intervalos de tempo diferentes e sob diferentes condições. Portanto, teste de software embarcado deve ser o mais automatizado possível, pois o custo e a efetividade do teste são cruciais para manter o produto competitivo frente ao dinamismo do mercado.

Assim, neste capítulo é apresentado o levantamento de trabalhos relacionados a este projeto de mestrado. Estes estudos apresentam em comum a proposição de critérios de teste para

robôs móveis, apresentam alguma estratégia de teste de integração para aplicações robóticas ou aplicações embarcadas compostas por diversos componentes. Desta maneira, na Seção 3.2 é apresentada uma síntese dos trabalhos encontrados durante o levantamento dos trabalhos relacionados, na Seção 3.3 são apresentados resumos dos estudos que mais se relacionam a este projeto de mestrado e na Seção 3.4 são apresentadas as considerações finais deste capítulo.

3.2 Contextualização

Na literatura são encontrados estudos que apresentam abordagens de teste para sistemas robóticos. Entretanto, não são encontradas abordagens que citam a definição de critérios de teste de integração para sistemas robóticos móveis. Neste contexto, há iniciativas para definição de critérios no estudo apresentado por Brito *et al.* (2015).

São encontrados alguns trabalhos que citam abordagens de teste de integração para sistemas robóticos compostos por diversos componentes (LIM *et al.*, 2009; KANG *et al.*, 2010; KANG; PARK, 2011; LIER F., 2013). Estas abordagens usam técnicas de geração de casos e de dados de teste, diferentes metodologias e ambientes para aplicação dos testes. Há alguns estudos que apresentam propostas de critérios de teste para sistemas robóticos, entretanto, não são realizados durante o teste de integração. Um exemplo é o estudo apresentado por Lill e Saglietti (2012) que apresentam critérios de teste baseados em redes de Petri para sistemas robóticos. Em relação ao uso de frameworks de desenvolvimento, em muitos trabalhos, é apresentado o uso do OPRos (*Open Platform for Robotic Services*) para aplicação de estudo de caso. Neves (2015) apresentam o uso do ROS, porém, este trabalho é voltado para o teste estrutural de sistemas robóticos.

Para sistemas embarcados são encontrados diversos trabalhos que apresentaram a proposição de critérios de teste de integração para sistemas embarcados (HOSSAIN; LEE, 2013; MATTIELLO-FRANCISCO *et al.*, 2012; BELLI; HOLLMANN; PADBERG, 2009; LEE; KIM; CHA, 2002; SEO *et al.*, 2007; SEO *et al.*, 2008; SUNG *et al.*, 2011). A maioria destes estudos apresentam critérios no nível de integração entre unidades que compõem estes sistemas como classes, métodos, funções, etc. Estes estudos não serão detalhados, devido ao objetivo principal deste trabalho ser direcionado a aplicações robóticas.

3.3 Teste de Software Aplicado a Sistemas Embarcados Robóticos

Lim *et al.* (2009) apresentam um modelo de teste hierárquico e um framework de teste para sistemas robóticos baseados em componentes de RTC (*Robot Technology Component*), em que é combinado com o módulo de hardware. Este modelo de teste hierárquico consiste de três níveis de testes baseados no V-model, em que são realizados testes unitários, teste de

integração e teste de sistema. O framework possui quatro módulos que são: um módulo para geração de dados de teste, um para gerenciamento dos testes, um módulo responsável pela execução dos testes e um módulo que realiza a monitoração dos testes. O propósito do modelo é ser eficiente para teste de componentes de software desenvolvidos para robôs em termo de custo e tempo. Assim, o modelo é utilizado para o desenvolvimento dos robôs desenvolvidos com RTC e com o framework OPRoS. A abordagem é composta por uma plataforma de software, APIs de hardware, um módulo de hardware e um simulador. A Figura 6 apresenta uma visão geral da abordagem. Por meio da figura é possível observar em qual nível é realizado cada teste e qual módulo está presente em cada nível de teste. O estudo apresenta experimentos realizados, em

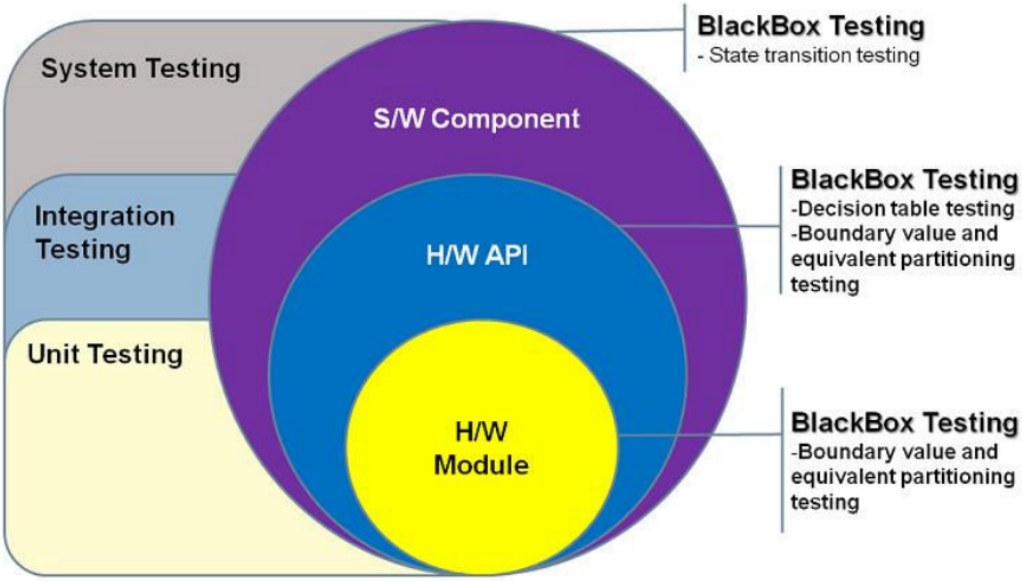

Figura 6 – Visão geral do modelo de teste (LIM *et al.*, 2009)

que demonstram a efetividade do modelo de teste proposto em termos de tempo, custo e também pela geração de vários casos de teste.

Kang *et al.* (2010) apresentam a ferramenta SITAT (*simulation-based interface testing automation tool*). Esta ferramenta permite a geração de casos de teste para o teste de interface de componentes de robôs móveis e também a execução da interface com os casos de teste gerados. Os casos de teste são gerados com base na especificação do sistema. Para execução dos casos de teste, ao invés da execução em módulo real de hardware é utilizado um simulador. SITAT gera e reduz o número de casos de teste para o teste de interface de software (S/W) de robôs e também executa o teste de interface no módulo de hardware (H/W). Para realizar a geração dos casos de teste a ferramenta possui uma arquitetura com 3 módulos, que são: *Web-based Interface Testing Automation Engine Server* (ITAE), *Test Build Agent* (TBA) e um *Robot H/W Simulator*. O módulo ITAE é a chave para a automação do processo de teste e tem sua base no gráfico de uso da interface. O ITAE é responsável pela geração dos casos de teste e geração de código fonte para o teste de aplicações como driver de teste, stubs e componentes de simulação para permitir a simulação dos componentes de hardware. O TBA é um agente responsável por gerenciar os

casos de teste e pode receber requisições automáticas de gerenciadores de teste para iniciar a execução dos testes. Este módulo é responsável pela aplicação dos casos de teste e apresentação dos resultados. O TBA é composto por um agente executor e por um compilador automático para execução dos casos de teste. O módulo simulador do hardware é implementado com as mesmas características do modelo real do hardware e é conectado aos TBAs. O estudo apresenta um exemplo de implementação, realizada com o framework OPRos. O teste de interface realizado é referente a interface do método de distância do robô e dos obstáculos inseridos no ambiente.

Kang e Park (2011) apresentam RPIST, uma metodologia para geração de casos de teste para interface, em que esta é aplicada a componentes de software de robôs. RPIST é composta por dois algoritmos principais. O primeiro algoritmo é utilizado para geração de casos de teste considerando a dependência entre as interfaces. O segundo disponibiliza uma arquitetura para execução dos casos de teste gerados. O algoritmo para geração de casos de teste é gerado com base em técnicas de geração de casos de teste baseadas em especificação. O algoritmo para geração dos casos de teste é composto por vários passos, em que as técnicas são combinadas. A arquitetura que permite a execução dos casos de teste, avalia o teste por meio da interação que ocorre do componente com um stub. A arquitetura de execução é composta por 3 componentes: (1)*Test Driver Component* (TD) que controla toda a operação de teste; (2)*Test Stub Component* (TS component) que disponibiliza uma interface virtual como a do componente que será testado e simula seu comportamento; e (3) utiliza-se de um arquivo com os casos de teste. A abordagem foi aplicada a um estudo de caso utilizando o framework OPRoS, em que foi testado um componente para robôs móveis chamado *'NavigationTO'*.

Belli, Hollmann e Padberg (2009) introduzem uma abordagem para o teste de integração de sistemas embarcados por meio de um Grafo de Sequência de Comunicação (GSC). O GSC descreve a comunicação entre os componentes de software representando mensagens como tuplas de parâmetros e variáveis globais que podem ser transmitidas por meio de uma forma válida ou inválida. A abordagem foca na comunicação entre componentes de software, enfatizando o fluxo de controle que deve ser integrado ao fluxo de dados. A geração de casos de teste do GSC é realizada com base em grafos de sequência de eventos, em que estes grafos são compostos por nós (eventos) e arcos que definem sequências de eventos direcionadas. A geração de casos de teste tem como objetivo cobrir todas as sequências de eventos de um dado tamanho. Com base no GSC foram definidos nove operadores básicos de mutação, que fazem a inserção ou exclusão de nós e arestas no GSC. O uso do teste de mutação é realizado para avaliação da adequação dos conjuntos de teste gerados. Os conjuntos de teste gerados com base no grafo de sequência de eventos são aplicados aos grafos mutantes.

Lill e Saglietti (2012) apresentam critérios de teste para sistemas robóticos móveis, onde estes critérios são baseados em redes de Petri. Redes de Petri é uma técnica de modelagem que permite a representação de sistemas, utilizando como alicerce uma forte base matemática. Essa técnica possui a particularidade de permitir modelar sistemas paralelos, concorrentes, assíncronos

e não-determinísticos (FRANCES, 2000). Os critérios de teste foram divididos em 3 categorias que são: Critérios de cobertura baseados em transição, critérios de cobertura baseados em eventos e critérios de cobertura baseados em estado. Os critérios definidos foram: *all transitions, all transition pairs, all transition sequences, all events, all event pairs, all event sequences, all states, all state pairs, all state sequences*. O estudo apresenta a aplicação dos critérios a um robô, o qual foi modelado usando ferramentas de CPN (*Coloured Petri Nets*). O estudo é expandido por Lill e Saglietti (2013), em que os critérios são incorporados a um modelo de teste focado no teste estrutural baseado nas redes de Petri.

Neves (2015) apresentam uma abordagem que automatiza o teste estrutural de software de veículos autônomos para apoio ao teste de campo. A abordagem é composta um modelo de classes no contexto de teste de campo, uma ferramenta que implementa esse modelo e um algoritmo genético para geração de dados de teste. De forma complementar, o estudo apresenta heurísticas para reduzir o conjunto de dados contidos contidos em um log sem diminuir a cobertura obtida e estratégias de combinação e mutação que são usadas no algoritmo. Os estudos de casos apresentados são realizados em veículos autônomos, desenvolvidos com o Framework ROS.

Estes estudos não foram considerados para o desenvolvimento deste projeto de mestrado. Este projeto de mestrado está focado no teste de integração entre os componentes de um sistema robótico, considerando a adaptação dos critérios de teste de programas concorrentes. Com isso, observou-se que os trabalhos relacionados não apresentam critérios de teste a nível de integração, em que o foco seja a comunicação dos componentes do sistema robótico. Outra limitação dos trabalhos encontrados é devido aos mesmos não apresentarem o uso do paradigma *publish/subscribe*, o qual é explorado por este trabalho.

3.4 Considerações Finais

Neste capítulo foi apresentado uma visão geral dos trabalhos que possuem algum tipo de relação com este projeto de mestrado. Foram apresentados também alguns critérios de teste propostos para robôs móveis. Não foram encontrados trabalhos que citam a definição ou aplicação de critérios de teste de integração entre os componentes de aplicações robóticas móveis. Poucos foram os frameworks de desenvolvimento utilizados pelos estudos e não foram encontrados trabalhos que citem a utilização do ambiente de desenvolvimento ROS para o teste de integração entre os componentes de sistemas robóticos. No próximo capítulo será apresentado um estudo da aplicação dos critérios de teste definidos para programas concorrentes para o teste de integração de robôs móveis.

CAPÍTULO

4

TESTE DE PROGRAMAS CONCORRENTES APLICADO A SISTEMAS ROBÓTICOS MÓVEIS

4.1 Considerações Iniciais

Conforme comentado anteriormente, sistemas robóticos móveis apresentam algumas características similares àquelas encontradas em softwares concorrentes, como comunicação entre os componentes, troca de mensagens e comportamento não determinístico. Essas características motivaram um estudo inicial para avaliar o quanto que o teste proposto para programas concorrentes é adequado para essas aplicações. Dessa forma, este capítulo apresenta um estudo de caso no qual o modelo e critérios de teste para programas concorrentes foram aplicados em um sistema robótico móvel. Para isso, os conceitos do modelo de teste foram mapeados para os conceitos envolvidos na aplicação de modo a identificar as possibilidades e limitações do teste definido para programas concorrentes para o teste de integração de sistemas robóticos móveis. Ressalta-se que esse mapeamento é um resultado obtido em conjunto com um trabalho de doutoramento (resultados parciais em (BRITO *et al.*, 2015)) no qual é proposto um modelo de teste e critérios de teste para aplicações robóticas no ROS.

Os conceitos apresentados foram definidos e detalhados na Seção 2.4. O capítulo está organizado da seguinte forma: na Seção 4.2 são apresentadas as adaptações realizadas para viabilizar a aplicação dos critérios de teste de programas concorrentes, para o contexto de robôs móveis. Na Seção 4.3.1 é apresentado a aplicação robótica explorada para realização deste estudo. Na Seção 4.3.2.1 é apresentado um estudo realizado, onde é apresentado como os critérios de teste de programas concorrentes as aplicações robóticas. Na Seção 4.4 são apresentadas discussões e análise sobre estudo, destacando as dificuldades e a importância de sua realização. Na Seção 4.5, são apresentadas as considerações finais deste capítulo.

4.2 Mapeamento dos Conceitos do Modelo de Teste

Esta seção apresenta o mapeamento dos conceitos de teste de programas concorrentes, para o teste de integração de sistemas robóticos móveis. Este mapeamento foi conduzido com base no estudo estudo do modelo de teste para programas concorrentes proposto por (SOUZA; VERGILIO; SOUZA, 2007) e no estudo das características dos sistemas robóticos. Devido às características do sistema robótico, optou-se por considerar o modelo de teste definido para programas concorrentes que utilizam comunicação por meio de passagem de mensagens, o qual foi descrito na Seção 2.4. A primeira atividade foi identificar os principais conceitos para o modelo de teste para programas concorrentes. Esses conceitos foram extraídos da definição (Seção 2.4) e consideram também o modelo de comunicação de aplicações robóticas desenvolvidas utilizando o framework ROS. Neste sentido, o mapeamento considera as características disponíveis no framework ROS.

Para o mapeamento dos conceitos é de suma importância o entendimento do modelo de comunicação *Publish/subscribe*. Neste modelo de comunicação há um conjunto de processos distribuídos que se registram (*subscribes*) para notificações de eventos de um determinado tipo. Quando um processo (*publish*) publica um evento, todos os processos que registraram seu interesse nesse tipo de evento são notificados. No momento em que um dado é publicado em um tópico, os processos que subscrevem aquele *Tópico*, no geral, possuem uma função *callback* que monitora a publicação de mensagens. Em passagem de mensagens (MPI) quando uma mensagem é enviada por meio de um *send* a mensagem que é enviada é consumida pelo processo. No modelo *publish/subscribe* do ROS a mensagem é recebida por um *subscribe* e ela não é consumida, pois a mesma é apenas lida do tópico, assim, caso possua mais de um *subscribe*, todos poderão ler a mensagem publicada no *tópico* e a mensagem somente será sobrescrita quando novas mensagens são recebidas. A função *callback* é similar ao *receive* não bloqueante. Uma rotina de comunicação é não bloqueante, quando a finalização da execução da rotina, não depende de certos eventos, ou seja, não há espera por uma ação, o processo continua sendo executado normalmente. Entretanto, a semântica do *receive* bloqueante é diferente a função do *callback*. Assim, o publish é similar ao *send* assincrono, o qual não espera o *receive* executar antes de continuar. Mas o conceito não é extensível para o *receive* síncrono. Neste estudo, assume-se que uma comunicação realizada no ROS é síncrona quando um processo aguarda que o respectivo envio do dado seja executado. Na comunicação assíncrona, por sua vez, o *publish* não necessita esperar um *subscribe* executar e ler os dados publicados para retornar. Da mesma forma, assume-se neste trabalho que um *publish* é bloqueante se ele garantir que o *buffer* com a mensagem (no espaço do usuário) pode ser reutilizado logo após a liberação do *tópico*, sem que a mensagem seja perdida ou sobreposta. Isso garante que o conteúdo da mensagem não seja corrompido pelo próprio processo transmissor após o *publish*.

No modelo *publish/subscribe* um processo é um nó do sistema distribuído, comunicando-se com outros processos por meio de passagem de mensagem, sendo produtores ou assinantes, ou

por meio de serviços, podendo agir como cliente ou servidor. Mensagens são tipos de dados em assinaturas e publicações em um canal de comunicação. Tópicos são canais de comunicação para publicação e assinatura de mensagens. Um processo Mestre (*Master*) funciona como um serviço de nomes e gerencia os demais processos durante a execução da aplicação. Nesse caso, como os processos possuem espaços de endereçamento distintos, a comunicação é feita pela troca (ou passagem) de mensagens. Esta ocorre de forma síncrona e bloqueante até que as mensagens estejam serializadas em um *buffer* para o transporte de cada *subscribe*. A escrita dos dados serializados para cada conexão *subscribe* ocorre a partir de diferentes *threads* do ROS, não disponíveis no espaço do usuário. Se os dados são publicados mais rápido do que a capacidade do *middleware* enviá-los pelo canal, mensagens mais antigas são descartadas.

Em relação à semântica dos paradigmas de comunicação e a ordem de envio e recebimento das mensagens, em passagem de mensagens MPI há garantias do envio e recebimento na ordem. No ROS, o envio e recebimento de mensagens se dá com a utilização do pacote de transmissão de mensagem TCP. Assim, há garantias que o pacote chegará na ordem e que os pacotes serão enviados até que eles sejam recebidos. Em programas concorrentes MPI, existem duas maneiras para se notificar o recebimento de uma mensagem, que são (GRAMA *et al.*, 2003b; SNIR *et al.*, 1996):

- *MPI_IRECV*: identifica uma área na memória para ser utilizada como *buffer* de recebimento das mensagens. O programa segue o processamento imediatamente após o recebimento da mensagem sem esperar que a mensagem recebida seja copiada para o buffer do aplicativo. A instrução de comunicação devolve uma mensagem com status pendente. O programa deve utilizar as chamadas *MPI_Wait* ou *MPI_Test* para determinar se a operação foi concluída.

- *MPI_Wait*: fica em estado de espera bloqueante até que a operação seja concluída. Para o caso de várias operações bloqueantes, o programador pode especificar os parâmetros ”Nenhum, alguns ou todos”.

O recebimento de uma mensagem em sistemas robóticos móveis é notificada por meio da função *callback*, o qual é utilizada para monitorar a publicação de mensagens em um tópico. Com essas configurações apresentadas o paradigma *Publish/subscribe* facilita a interação entre os processos de um sistema distribuído, por prover fraco acoplamento, resultante de três características principais:

1. O serviço de eventos é o intermediário na comunicação entre os processos que publicam e os que subscrevem eventos, de forma que o processo assinante não tem conhecimento sobre quem publicou determinado evento;

2. Um evento é propagado de forma assíncrona para todos os *subscribes* que registraram interesse naquele evento, de forma que o *subscribe* não precisa estar ativo no momento que o evento foi publicado; e

3. *Publishes* não são bloqueados enquanto produzem eventos e *subscribes* podem ser notificados de forma assíncrona (por meio de *callbacks*) da ocorrência de um evento enquanto estão realizando alguma atividade concorrente.

Entre as principais semelhanças encontradas na semântica de uma aplicação robótica para programas concorrentes é com relação a execução paralela. Uma aplicação concorrente é composta por processos que são executados paralelamente e em aplicações robóticas é similar a execução paralelas dos Nós (componentes). Outra característica similar é relacionada à comunicação. Durante a execução de um programa concorrente os processos precisam interagir para troca de mensagens entre si, para executar uma tarefa. Um sistema robótico móvel é composto por diversos Nós como sensores e atuadores que são executados paralelamente. A comunicação e sincronização entre as tarefas desempenhadas por cada um dos dispositivos, muitas vezes, é não determinística, assim como em programas concorrentes, em que há diversos processos/*threads* executando de forma concorrente.

Em programas concorrentes, diz-se que um programa é concorrente, quando dois ou mais processos tiveram iniciado sua execução, mas não tiverem terminado (SOUZA; VERGILIO; SOUZA, 2007). Em aplicações robóticas, este conceito pode ser mapeado. Diz-se que dois componentes são concorrentes, quando em instante iniciaram sua execução e não tiveram terminado.

4.3 Aplicação dos critérios de teste

Esta seção apresenta a aplicação dos conceitos apresentados e mapeados, demonstrando como são aplicados os critérios de teste de programas concorrentes a uma aplicação robótica móvel.

4.3.1 Estudo de caso: Robô iRobot Roomba

Esta seção apresenta o programa robótico móvel (aspirador de pó), que é uma versão simplificada do iRobot Roomba[1]. Este programa foi desenvolvido com base em uma parte do código disponibilizada por Lucas, Elbaum e Rosenblum (2012). Com base no código disponibilizado, o código do componente *controller* foi implementado, permitindo analisar com maiores detalhes a comunicação entre os componentes. A Figura 7 ilustra um grafo gerado pelo *ROS* para este programa composto por processos produtores e assinantes de conteúdo conectados por tópicos.

[1] Irobot - http://www.irobot.com/Company

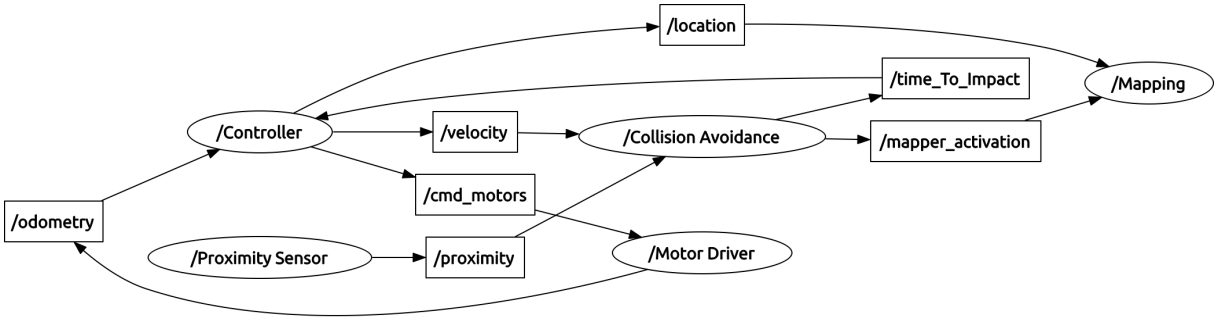

Figura 7 – Grafo gerado pelo ROS para o robô aspirador.

O programa contém 5 processos: *Controller*, *Proximity Sensor*, *Collision Avoidance*, *Motor Driver* e *Mapping*. A comunicação entre esses processos é estabelecida usando os seguintes tópicos: *odometry*, *proximity*, *cmd_motors*, *velocity*, *location*, *time_To_Impact* e *mapper_activation*. As listagens 5 e 6 descrevem duas partes do código resumido da aplicação relacionadas aos componentes *Collision Avoidance* (LUCAS; ELBAUM; ROSENBLUM, 2012) e *Controller* (BRITO *et al.*, 2015). O componente *Controller* recebe informações do tópico *time_To_Impact* (linha 11) publicado pelo componente *Collision Avoidance* e processa-as. No próximo passo ele publica dados no tópico *cmd_motors* (linha 29), que são assinados pelo componente *Motor Driver*. O componente *Controller* também publica a velocidade do robô no tópico *velocity* (linha 28), que é assinada pelo componente *Collision Avoidance* (line 8). Cada componente possui duas funções *callback()* que contém código relacionado a mensagens que chegam em cada nó. Por exemplo, se o componente *Controller* publica dados no tópico *velocity*, a função chamada *velocityCallback()* é ativada (linha 8). O componente *Collision Avoidance* será notificado se um nó tem dados publicados de seu interesse, porque ele é assinante do tópico *velocity*. A função *time_To_ImpactCallback()* é invocada quando dados são publicados no tópico *time_To_Impact*.

Código-fonte 5: Resumo do código do componente *Collision Avoidance* (LUCAS; ELBAUM; ROSENBLUM, 2012) component.

```
1  //Collision Avoidance Component
2  State State; // {working, slowed, stopped, crashed}
3  int proximity;
4
5  proximityCallback(int proximityP) {
6    proximity = proximityP;
7
8  velocityCallback(int speed) {
9    if (speed == 0)
10     time_to_impact = 9999;
11   else
12     time_to_impact = proximity / speed;
13   if (time_to_impact < 2) {
14     if (state != working)
15       state = stopped;
16     else {
17       state = crashed;
```

```
18        assert(false); //Failure
19      }
20    }
21    else
22      if (time_to_impact < 3) {
23        state = slowed;
24        reduceMapping();
25      }
26      else {
27        state = working;
28        activeMapping();
29      }
30    publish(time_to_impact);
31  }
```

Código-fonte 6: Resumo do código do componente *Controller*

```
1  //Controller Component
2  State state; //{working, slowed, stopped, crashed}
3  int time_to_impact;
4  int speed; //{0, 1, 2}
5  int c_motors; //{stop, reduce, working}
6
7  odometryCallback(int odometryX) {
8      odometry = odometryX;
9  }
10
11 time_to_impactCallback(int timeToImpactT) {
12     time_to_impact = timeToImpactT;
13     if (time_to_impact < 2) {
14         c_motors = stop;
15         speed = 0; //break
16         state = crashed;
17         activeMapping();
18     } else if (time_to_impact <= 3) {
19         c_motors = reduce;
20         speed = 1;
21         reduceMapping();
22         state = slowed;
23     } else {
24         c_motors = working;
25         speed = 2;
26         state = working;
27     }
28     publish(speed);
29     publish(c_motors);}
```

Essa aplicação apresenta as principais características de comunicação do modelo ROS e pela simplicidade foi escolhida para o estudo apresentado na sequência.

4.3.2 Aplicação dos critérios de teste para programas concorrentes

Esta seção apresenta um exemplo de aplicação dos critérios de teste de programas concorrentes a aplicação robótica Irobot Roomba apresentada na Seção 4.3.1. Para este estudo

são considerados critérios das seguintes famílias de critérios baseados em fluxo de controle e no fluxo de comunicação e Critérios baseados em fluxo de dados e em passagem de mensagens (SOUZA; VERGILIO; SOUZA, 2007). Alguns critérios de teste de programas concorrentes não foram considerados para este estudo. Isso se deve ao fato de que este estudo está voltado ao teste de integração de aplicações robóticas e os critérios de teste de programas concorrentes foram definidos para o teste estrutural. Devido ao objetivo deste trabalho, o foco está voltado aos critérios baseados no fluxo de comunicação e em passagens de mensagens. Exemplo de critérios não utilizados são os critérios Todos-p-usos, Todos-c-usos, Todos-Nós, Todas-Arestas e Todas-Defs, estes critérios tem por objetivo coletar informações internas do processo e não a informações inerentes à comunicação destes sistemas

Para aplicação dos critérios, o primeiro passo é executar a aplicação robótica a ser testada. Com a execução da aplicação robótica utilizando o framework ROS é obtido o seu grafo de componentes. Para gerar este grafo é preciso que a aplicação seja executada e todos os componentes que se deseja testar estejam em execução. A partir do grafo de componentes é gerado um grafo similar ao GFCP e por meio desse grafo são extraídas da aplicação robótica informações importantes para aplicação dos critérios. A partir destas informações são obtidos os elementos requeridos. O próximo passo é gerar os casos de teste que sejam capazes de cobrir os elementos requeridos. Em aplicações robóticas os casos de teste podem ser de diversos tipos de dados e vindos de diversos componentes como sensores, gps, câmera, dentre diversos outros. A partir de então, a aplicação é executada com o conjunto de casos de teste.

Com base no grafo de comunicação apresentado na Figura 7, a aplicação foi mapeada para um grafo similar ao GFCP, apresentado na Figura 8.

Neste grafo há algumas diferenças com relação a representação do GFCP. Algumas adaptações foram necessárias, pois o mesmo é baseado no fluxo de comunicação *Publish/Subscribe* e foram aplicados os conceitos mapeados e apresentados na Tabela 1. Com isso, no grafo os processos são representados como retângulos, interligados por arestas interprocessos. Cada retângulo é composto por $CFGs$ que representam métodos desse processo. As arestas sólidas representam as arestas *intramétodos* ($E^{p,m}$) e as arestas tracejadas representam as arestas *interprocessos* (E_i). No grafo, cada processo p^i é composto por um número nm fixo e conhecido de métodos, definido por $p^i = m^{i,0}, m^{i,1}, ...m^{i,(nm^{i-1})}$. Cada método m^p possui seu $GFC^{p,m}$, construído utilizando-se os mesmos conceitos do GFC de programas tradicionais. O $GFC^{p,m}$ é composto de um conjunto de nós $N^{p,m}$ e de um conjunto de arestas $E^{p,m}$. A notação $n_i^{p,m}$ é utilizada para representar cada nó do $GFC^{p,m}$, sendo que n corresponde ao nó, i é a identificação do nó, p indica o processo e m o método. O conjunto de arestas entre os métodos de p é denominado (E^p). N^p representa o conjunto de todos os nós de p.

A partir destas informações, na Tabela 1 são apresentadas as principais informações relacionadas a sistemas robóticos, onde são apresentadas os conceitos mapeados de programas concorrentes. Na primeira coluna é apresentado o conceito definido para programa concorrente e

na segunda é apresentado o seu mapeamento para o contexto de sistemas robóticos móveis.

Tabela 1 – Mapeamento dos conceitos do modelo de teste de programas concorrentes para sistemas robóticos.

Programas Concorrentes	**Sistemas Robóticos**
Conjunto de Processos	Conjunto de Componentes (Nós)
Comunicação Send/receive	Comunicação Publish/ Subscribe
Nós Intraprocesso (ni)	Nós Intracomponente
Arestas de comunicação (Es)	Tópicos
Arestas intraprocesso (Ei)	Arestas intracomponentes (Ei)
Caminho intraprocesso	Caminho intracomponente
Caminho interprocessos	Caminho intercomponente
uso computacional (c-uso)	uso computacional (c-uso)
uso predicativo (p-uso)	uso predicativo (p-uso)
uso comunicacional (m-uso)	uso comunicacional (m-uso)

A Tabela 2 contém os conjuntos obtidos a partir das definições no grafo e são utilizados para geração dos requisitos de teste. Estes conjuntos foram definidos a partir das informações mapeadas entre aplicações concorrentes e sistemas robóticos. Em seguida, a Tabela 3 apresenta o conjunto de variáveis definidas.

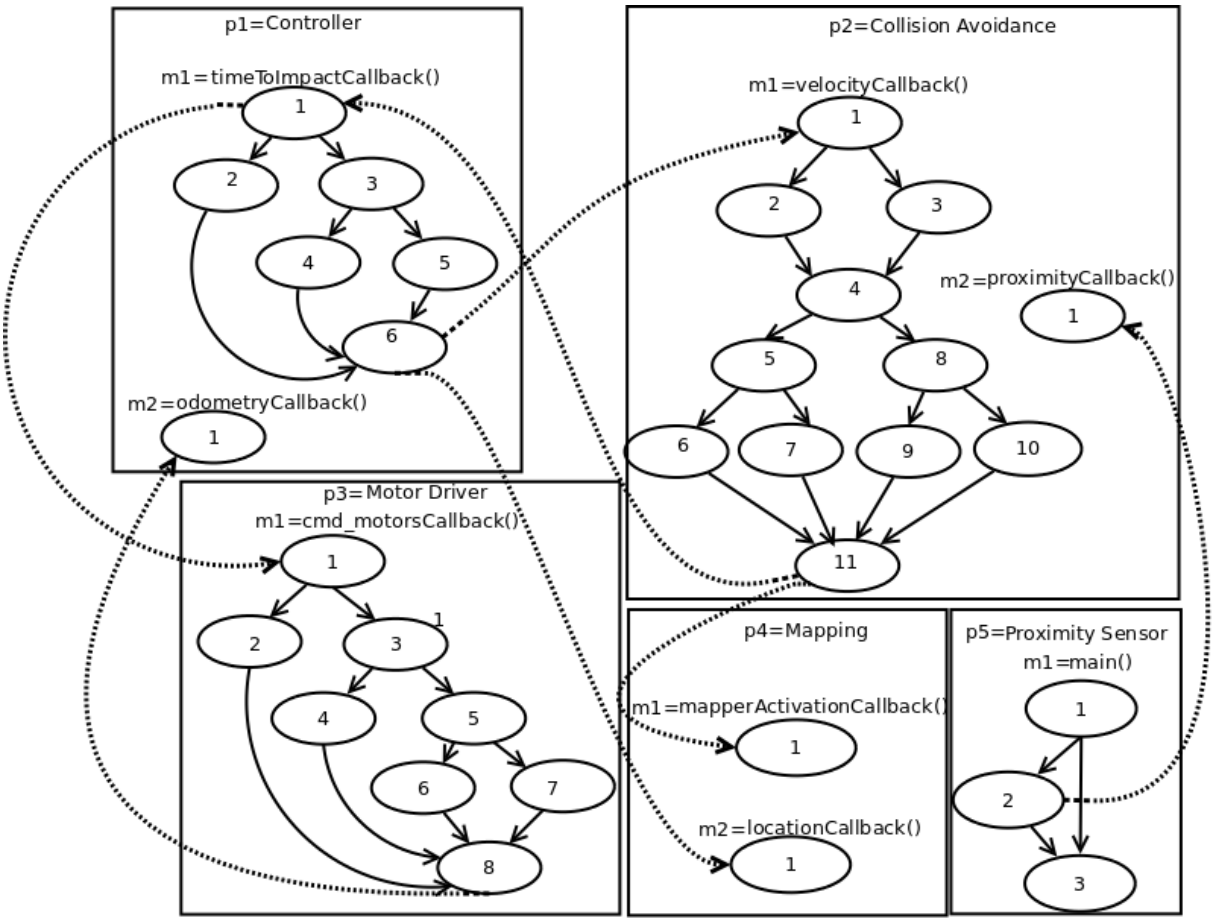

Figura 8 – GFCP para o robô aspirador.

Tabela 2 – Informações obtidas pelo modelo de teste para o robô aspirador de pó.

$np = 5,$ $Prog = \{p^1, p^3, p^4, p^5\}$
$p^1 = \{m^1, m^2\}$ $p^2 = \{m^1, m^2\}$ $p^3 = \{m^1\}$ $p^4 = \{m^1, m^2\}$ $p^5 = \{m^1\}$
$N^{1,1} = \{n_1^{1,1} ... n_6^{1,1}\}$ $N^{1,2} = \{n_1^{1,2}$ $N^{2,1} = \{n_1^{2,1}, ..., n_{11}^{2,1}\}$ $N^{2,2} = \{n_1^{2,2}\}$ $N^{3,1} = \{n_1^{3,1} ... n_8^{3,1}\}$ $N^{4,1} = \{n_1^{4,1}\}$ $N^{4,2} = \{n_1^{4,2}\}$ $N^{5,1} = \{n_1^{5,1}, .., n_3^{5,1}\}$
$N^1 = N^{1,1} \cup N^{1,2}$ $N^2 = N^{2,1} \cup N^{2,2}$ $N^3 = N^{3,1}$ $N^4 = N^{4,1} \cup N^{4,2}$ $N^5 = N^{5,1}$ $N = N^1 \cup N^2 \cup N^3 \cup N^4 \cup N^5$
D^1 = { *(d1,'odometry'), (d2,'time_to_impact'), (d3, 'c_motors'), (d4, 'c_motors'), (d5, 'c_motors'), (d6, 'speed'), (d7, 'speed'), (d8, 'speed'), (d9, 'state'), (d10, 'state'), (d11, 'state')* } D^2 = { *(d12, 'proximity'), (d13, 'time_to_impact'), (d14, 'time_to_impact'), (d15, 'state'), (d16, 'state'), (d17, 'state'), (d18, 'state')* } D^3 = { *(d19,'msg')* } D^4 = { *(d20, 'map'), (d21, 'pose')* } D^5 = { *(d22,'n'), (d23,'pub'), (d24, 'msg')* }
$E^{1,1} = \{(n_1^{1,1}, n_2^{1,1}), (n_2^{1,1}, n_6^{1,1}), (n_1^{1,1}, n_3^{1,1}), (n_3^{1,1}, n_4^{1,1}), (n_4^{1,1}, n_6^{1,1}), (n_3^{1,1}, n_5^{1,1})\}$ $\{(n_5^{1,1}, n_6^{1,1})\}$ $E^{2,1} = \{(n_1^{2,1}, n_2^{2,1}), (n_2^{2,1}, n_4^{2,1}), (n_4^{2,1}, n_5^{2,1}), (n_2^{2,1}, n_6^{2,1}), (n_6^{2,1}, n_1 1^{2,1}), (n_5^{2,1}, n_7^{2,1})\}$ $\{(n_7^{2,1}, n_1 1^{2,1}), (n_4^{2,1}, n_8^{2,1}), (n_8^{2,1}, n_9^{2,1}), (n_9^{2,1}, n_1 1^{2,1}), (n_8^{2,1}, n_1 0^{2,1}), (n_1 0^{2,1}, n_1 1^{2,1}), \}$ $\{(n_1^{2,1}, n_3^{2,1}), (n_3^{2,1}, n_4^{2,1})\}$ $E^{3,1} = \{(n_1^{3,1}, n_2^{3,1}), (n_2^{3,1}, n_8^{3,1}), (n_1^{3,1}, n_3^{3,1}), (n_3^{3,1}, n_4^{3,1}), (n_4^{3,1}, n_8^{3,1}), (n_3^{3,1}, n_5^{3,1}), \}$ $\{(n_5^{3,1}, n_6^{3,1}), (n_6^{3,1}, n_8^{3,1}), (n_5^{3,1}, n_7^{3,1}), (n_7^{3,1}, n_8^{3,1})\}$ $E_s = \{(n_1^{1,1}, n_1^{3,1}), (n_6^{1,1}, n_1^{4,2}), (n_{11}^{2,1}, n_1^{4,1}), (n_8^{3,1}, n_1^{1,1}), (n_1^{5,1}, n_1^{2,2})$ $E = E^{1,1} \cup E^{2,1} \cup E^{3,1} \cup E_s$

Tabela 3 – Variáveis definidas para o robô aspirador de pó.

defs:	$\langle n_1^{1,2}, d1\rangle$, $\langle n_1^{1,1}, d2\rangle$, $\langle n_2^{1,1}, d3\rangle$, $\langle n_4^{1,1}, d4\rangle$, $\langle n_5^{1,1}, d5\rangle$, $\langle n_2^{1,1}, d6\rangle$, $\langle n_4^{1,1}, d7\rangle$, $\langle n_5^{1,1}, d8\rangle$, $\langle n_2^{1,1}, d9\rangle$, $\langle n_4^{1,1}, d10\rangle$, $\langle n_5^{1,1}, d11\rangle$, $\langle n_1^{2,2}, d12\rangle$, $\langle n_2^{2,1}, d13\rangle$, $\langle n_3^{2,1}, d14\rangle$, $\langle n_6^{2,1}, d15\rangle$, $\langle n_7^{2,1}, d16\rangle$, $\langle n_9^{2,1}, d17\rangle$, $\langle n_{10}^{2,1}, d18\rangle$, $\langle n_1^{3,1}, d19\rangle$, $\langle n_1^{4,1}, d20\rangle$, $\langle n_1^{4,2}, d21\rangle$, $\langle n_1^{5,1}, d22\rangle$, $\langle n_1^{5,1}, d23\rangle$, $\langle n_3^{5,1}, d24\rangle$,

A Tabela 4 apresenta exemplos de elementos requeridos para o programa da Figura 8. A geração dos elementos requeridos foi realizada com base no mapeamento das informações entre as aplicações concorrentes e robóticas. Um exemplo para os critérios baseados no fluxo de comunicação é o critério Todos-Nós-r. Em programas concorrentes esse critério é baseado no fluxo de comunicação e requer que todos os nós que possuem um *receive* sejam exercitados. Neste exemplo, a aplicação deste critério a aplicação robótica é definida de maneira que todos os nós que possuem *subscribe* sejam exercitados. Um exemplo desta adaptação e aplicação é apresentado pelo elemento requerido: $(n_1^{1,1})$, em que neste nó há um evento de um *subscribe*. Da mesma forma ocorre com o critério Todos-Nós-s. Para este critério, em programas concorrentes, todos os nós que possuem um *send* são considerados requisitos de teste. Na aplicação robótica, a definição é estendida para todos os nós que possuem um *publish*. Um exemplo de um elemento requerido para o critério Todos-Nós-s é: $(n_6^{1,1})$, onde neste nó há um evento de *publish*.

Em relação aos critérios baseado no fluxo de dados e em passagem de mensagens, pode-se exemplificar o critério Todos-s-Usos, em que este critério requer que todas as associações s-uso sejam exercitadas. Assim, um requisito de teste para este critério na aplicação robótica pode ser exemplificada por: $(n_4^{1,1}(n_6^{1,1}, n_1^{2,1}), speed)$. Este elemento requerido tem como informações a variável (*speed*) o nó em que esta variável foi definida $(n_4^{1,1})$, o nó em que houve um evento de *publish* para esta variável $((n_6^{1,1})$, o nó em que esta variável foi recebida por um evento *subscribe* de outro processo $(n_1^{2,1})$ em um caminho livre de definição.

Na Tabela 5 são apresentados os casos de teste gerados para essa aplicação. Uma entrada de teste para este programa é um dado a partir do componente *Proximity Sensor* e a saída esperada consiste em um comando do componente *Motor Driver* para os atuadores do robô (rodas). Os casos de teste foram gerados utilizando a técnica de teste funcional do particionamento de classes de equivalência. Durante a execução do teste, os dados de teste percorrem um conjunto de caminhos, conforme definido anteriormente. Cabe ressaltar que como ocorre em programas concorrentes, alguns requisitos não são executáveis e cabe ao testador analisar esses elementos.

4.3.2.1 Ferramentas de apoio a execução dos testes

Para auxiliar na aplicação dos casos de teste, foram utilizadas as ferramentas Rostest e Gtest. A Figura 9 apresenta um exemplo da configuração utilizada para o Rostest para este estudo. Na linha 2 está sendo chamado o primeiro nó de teste *proximity*, na linha 3 é indicado o

Tabela 4 – Elementos requeridos para os critérios para o robô aspirador de pó.

Critério	**Elementos requeridos**
Todas-Arestas-s	$(n_1^{1,1}, n_1^{3,1}), (n_6^{1,1}, n_1^{4,2}), (n_{11}^{2,1}, n_1^{4,1}), (n_8^{3,1}, n_1^{1,1})$, $(n_2^{5,1}, n_1^{2,2}), (n_6^{1,1}, n_1^{2,1}), (n_1^{2,1}, n_1^{1,1})$
Todos-Nós-r	$(n_1^{1,1})$, $(n_1^{2,1})$, $(n_1^{1,2})$, $(n_1^{2,2})$, $(n_1^{3,1})$, $(n_1^{4,1})$, $(n_1^{4,2})$
Todos-Nós-s	$(n_6^{1,1})$, $(n_{11}^{2,1})$, $(n_8^{3,1})$, $(n_2^{5,1})$
Todas-Defs/s	$(n_2^{1,1}(n_6^{1,1}, n_1^{2,1}), n_1^{2,1}, speed, speed)$, $(n_2^{1,1}(n_6^{1,1}, n_1^{3,1}), n_1^{3,1}, c_motor, c_motor)$, $(n_2^{2,1}(n_{11}^{2,1}, n_1^{1,1}), n_1^{1,1}, time_to_impact, time_to_impact)$, $(n_1^{1,1}(n_6^{1,1}, n_1^{4,2}), n_1^{4,2}, location, location)$, $(n_1^{2,1}(n_{11}^{2,1}, n_2^{4,1}), n_2^{4,1}, mapperActivation, mapperActivation)$, $(n_1^{5,1}(n_2^{5,1}, n_1^{2,2}), n_1^{2,2}, proximity, proximity)$, $(n_1^{3,1}(n_8^{3,1}, n_1^{2,2}), n_1^{2,2}, odometry, odometry)$,
Todos-s-Usos	$(n_4^{1,1}(n_6^{1,1}, n_1^{2,1}), speed)$, $(n_5^{1,1}(n_6^{1,1}, n_1^{2,1}), speed)$, $(n_4^{1,1}(n_6^{1,1}, n_1^{3,1}), c_motor)$, $(n_5^{1,1}(n_6^{1,1}, n_1^{3,1}), c_motor)$, $(n_2^{2,1}(n_{11}^{2,1}, n_1^{1,1}), time_to_impact)$, $(n_3^{2,1}(n_{11}^{2,1}, n_1^{1,1}), time_to_impact)$, $(n_1^{5,1}(n_2^{5,1}, n_1^{2,2}), proximity)$, $(n_1^{3,1}(n_8^{3,1}, n_1^{1,2}), odometry)$
Todos-s-c-Usos	$(n_4^{1,1}(n_6^{1,1}, n_1^{2,1}), n_1^{2,1}, speed, speed)$, $(n_4^{1,1}(n_6^{1,1}, n_1^{3,1}), n_1^{3,1}, c_motor, c_motor)$, $(n_3^{2,1}(n_1 1^{2,1}, n_1^{1,1}), n_1^{1,1}, time_to_impact, time_to_impact)$, $(n_1^{5,1}(n_2^{5,1}, n_1^{2,2}), n_1^{2,2}, proximity, proximity)$, $(n_1^{3,1}(n_8^{3,1}, n_1^{1,2}), n_1^{1,2}, odometry, odometry)$
Todos-s-p-Usos	$(n_4^{1,1}(n_6^{1,1}, n_1^{2,1}), n_1^{2,1}, speed, speed)$, $(n_4^{1,1}(n_6^{1,1}, n_1^{3,1}), n_2^{3,1}, c_motor, c_motor)$, $(n_3^{2,1}(n_1 1^{2,1}, n_1^{1,1}), n_2^{1,1}, time_to_impact, time_to_impact)$,

Tabela 5 – Casos de teste.

Entrada	Saída esperada
-1	stop
1	stop
2	reduce
3	work
100	work

pacote onde consta este nó, na linha 4 diz-se o tipo deste componente, na linha 5 indica que o resultado do teste deve ser apresentado em tela. Na linha 6 é apresentado o nome deste teste, na linha 7 o pacote onde consta os componentes, o tipo do teste na linha 8 e na linha 9 é definido o tempo limite para execução do teste, antes que seja considerado como falho. Nas demais linhas são apresentados os outros nós testados nesta interação, juntamente com suas configurações.

```
1 <launch>
2  <node name="proximity"
3         pkg="rostest_node_example"
4         type="proximity"
5         output="screen"/>
6  <test test-name="TestAspirador"
7         pkg="rostest_node_example"
8         type="test_proximity"
9         time-limit="60.0"/>
10 <node name="collision"
11        pkg="rostest_node_example"
12        type="collision"
13        output="screen"/>
14 <node name="controller"
15        pkg="rostest_node_example"
16        type="controller"
17        output="screen"/>
18 <node name="motor_drive"
19        pkg="rostest_node_example"
20        type="motor_drive"
21        output="screen"/>
22 <node name="mapping"
23        pkg="rostest_node_example"
24        type="mapping"
25        output="screen"/>
26 </launch>
```

Figura 9 – Exemplo de configuração do Rostest para execução dos testes.

O Código-fonte 7 apresenta um exemplo de um caso de teste programado utilizando o Gtest. Neste exemplo o caso de teste está localizado entre as linhas 72 à 77. Neste caso de teste a

saída esperada para o tópico *cmd_motors* é 3, caso contrário o teste o resultado do teste não será o esperado. Este caso de teste é executado com todos os componentes em execução, trocando informações entre seus tópicos.

Código-fonte 7: Exemplo de caso de teste executado.

```
1 #include "gtest/gtest.h"
2 #include "ros/ros.h"
3 #include "std_msgs/Int16.h"
4 #include "std_msgs/Int8.h"
5
6 class Callback : public testing::Test {
7  protected:
8   virtual void SetUp() {
9
10     proximity_sub = n.subscribe("proximity", 1000, &Callback::proximityCallback,
       this);
11     sub1 = n.subscribe("timeToImpact", 1000, &Callback::timeToImpactCallback, this);
12     velocity_sub = n.subscribe("velocity", 1000, &Callback::velocityCallback, this);
13     cmd_sub = n.subscribe("cmd_motors", 1000, &Callback::cmdCallback, this);
14
15      }
16
17   virtual void TearDown() {
18
19   }
20
21   ros::NodeHandle n;
22   ros::Subscriber sub1;
23   ros::Subscriber proximity_sub;
24   ros::Subscriber velocity_sub;
25   ros::Subscriber cmd_sub;
26
27   int msgProximity;
28   int msgVelocity;
29   int msgTimeToImpact;
30   int cmd_motor;
31  public:
32  void proximityCallback(const std_msgs::Int8::ConstPtr& msgP)
33 {
34     msgProximity = msgP->data;
35
36  }
37
38  void timeToImpactCallback(const std_msgs::Int8::ConstPtr& msgT){
39
40        msgTimeToImpact = msgT->data;
41
42   }
43
44  void velocityCallback(const std_msgs::Int8::ConstPtr& msgV)
45 {
46    msgVelocity = msgV->data;
47
48  }
49
```

```
50  void cmdCallback(const std_msgs::Int8::ConstPtr& msgC)
51 {
52   cmd_motor = msgC->data;
53 }
54
55   int get_Proximity(){
56       return msgProximity;
57   }
58
59   int get_timeToImpact(){
60       return msgTimeToImpact;
61   }
62
63  int get_Velocity(){
64       return msgVelocity;
65   }
66
67   int get_Cmd(){
68       return cmd_motor;
69   }
70 };
71
72 TEST(Callback, Test_Node_Motor_Drive) {
73    ros::Rate loop_rate(1);
74    loop_rate.sleep();
75    EXPECT_EQ(3, this->get_Cmd());
76
77 }
78 int main(int argc, char **argv) {
79   ::testing::InitGoogleTest(&argc, argv);
80   ros::init(argc, argv, "test_aspirador");
81   return RUN_ALL_TESTS();
82 }
```

Com a utilização do Rostest juntamente ao Gtest é possível automatizar a execução dos casos de teste para aplicação dos critérios. Estas ferramentas trabalhando em conjunto permitem a realização do teste e integração no ROS. Assim, ao aplicar os casos de teste pode-se definir quais processos estarão em execução, quais os dados de entrada e qual a saída esperada para o momento da execução dos processos. Com isso, é possível analisar como os processos se comportam trabalhando juntos e trocando informações por meio do modelo *publish/subscribe*.

No exemplo apresentado apresentado na Figura 9 e no Código-fonte 7 o teste foi realizado com todos os processos da aplicação em execução. A partir dos dados que eram passados como entrada para o componente *proximity Sensor*, valor esperado na saída do *tópico cmd_motors* seria 3, qualquer valor diferente de 3 o teste seria considerado falho. Este estudo contemplou estas ferramentas a fim de demonstrar que as mesmas são úteis para execução dos casos de teste definidos para o teste de integração a estas aplicações. Como trabalhos futuros sugere-se a execução e exploração de forma experimental para verificar a efetividade destas ferramentas neste contexto.

4.4 Discussões

O objetivo principal deste estudo foi demonstrar a aplicabilidade e adaptabilidade dos critérios de teste de programas concorrentes para o teste de integração de sistemas robóticos móveis. Outro objetivo do estudo foi mapear ferramentas que auxiliassem na automatização do processo de teste, assim, foram exploradas as ferramentas Rostest e Gtest.

Algumas dificuldades são encontradas no teste de sistemas robóticos móveis em relação a programas concorrentes e sequenciais. Uma das dificuldades é representar o programa para robô móvel e capturar as informações pertinentes à atividade de teste, conforme gerado em programas concorrentes. Outra dificuldade encontrada está relacionada com o objetivo do estudo, por que os critérios definidos para programas concorrentes são estabelecidos para o teste estrutural e no contexto deste estudo os critérios estão sendo utilizados para o teste de integração entre os diversos componentes que compõe uma aplicação robótica.

A realização deste estudo foi de suma importância para este projeto de mestrado. A aplicação apresentada e utilizada é considerada base para as definições e para o mapeamento das informações pertinentes para aplicação dos critérios de teste de programas concorrentes, pois apesar de simples esta aplicação contém todos os elementos e características necessárias para exemplificar a aplicação dos critérios.

Os resultados encontrados com a realização deste estudo, demonstram a similaridade entre aplicações de programas concorrentes e aplicações robóticas. A partir deste estudo, pode-se concluir que os critérios de teste definidos para programas concorrentes podem ser explorados para guiar a definição de teste de integração em aplicações robóticas.

4.5 Considerações Finais

Neste capítulo foi apresentado um estudo, em que os critérios de teste de programas concorrentes são aplicados no contexto de aplicações robóticas móveis. Foi apresentado o mapeamento dos conceitos de programas concorrentes para aplicações robóticas. Foi também apresentado um estudo onde os conceitos apresentados e mapeados são aplicados a uma aplicação robótica. No próximo capítulo será apresentado o estudo de caso 2 realizado com a aplicação real CaRINA Parking.

CAPÍTULO

5

ESTUDO DE CASO 2

5.1 Considerações Iniciais

Este capítulo apresenta um estudo de caso conduzido utilizando a aplicação robótica real CaRINA Parking (MATIAS *et al.*, 2015), o qual é uma versão da veículo autônomo CaRINA. Quando em execução o objetivo da aplicação é projetar o veículo de um ponto inicial até uma vaga de estacionamento provida pelo usuário.

Este capítulo está organizado de maneira que, na Seção 5.2 é apresentada a aplicação e uma visão geral de seu funcionamento. A Seção 5.3 apresenta a aplicação dos critefios de teste de programas concorrentes ao CaRINA Parking. Na Seção 5.3.1 é apresentada a execução dos testes. Na Seção 5.4 é apresentado a discussão e análise dos resultados obtidos com a realização deste estudo. Por fim, na Seção 5.5 são apresentadas as considerações finais deste capítulo.

5.2 A Aplicação

O CaRINA Parking é uma aplicação vinculada ao projeto CaRINA[1], em que possui como objeto o desenvolvimento de um módulo que será aclopado ao sistema de estacionamento do veículo autônomo. Para executar a aplicação o usuário fornece ao robô um vetor destino, representando o local da vaga de estacionamento que o robô deve alcançar e estacionar. Em seguida, o robô traça a rota que ele deve percorrer para chegar a esse objetivo, considerando o prévio conhecimento de obstáculos do mapa. Um exemplo do funcionamento da aplicação é apresentado na Figura 10. A aplicação é desenvolvida utilizando o framework ROS e o simulador Morse [2]. O Morse é um simulador para aplicações robóticas, que apresenta a simulação de aplicações robóticas em 3D para pequenos ou grandes ambientes.

[1] CaRINA - http://www.lrm.icmc.usp.br/carina/
[2] MORSE - www.openrobots.org/morse

A Figura 11 ilustra o grafo de comunicação gerado pelo ROS para esta aplicação. Ela é composta por 10 Nós: *Morse*, *link1*, *link2*, *link3*, *slamgmapping*, *map_server*, *amcl*, *Move_-base*, *ackermann_pub* e *ackermann_sub*. A comunicação entre esses nós é estabelecida usando os seguintes tópicos: *tf*, *scan*, *initialpose*, *odom*, *map*, *move_base_simple/goal*, *sick*, *move_-base/cancel*, *cmd_vel*, *move_base_goal*, *move_base/SBPLLatticePlanner/plan* e *ackermann_-cmd*.

Durante a execução da aplicação os componentes interagem entre si por meio da comunicação baseada em *publish/subscribe* ou serviços. Neste trabalho, como o foco está no paradigma publish/subscribe, não serão apresentados detalhes sobre a comunicação usando serviços. O simulador *morse* publica dados nos tópicos *sick* e *odom*. No tópico *sick* são publicados dados do sensor laser utilizados para identificar objetos próximos do veículo e auxiliar no cálculo da rota a ser traçada e do caminho a ser percorrido pelo robô. No tópico *odom* são publicados dados referentes a posição do robô no mapa. O Nó *amcl* é responsável por estimar a localização do robô no mapa e ao iniciar a aplicação subscreve o tópico *initialpose* para obter as coordenadas do robô no mapa. Os componentes *link1*, *link2* e *link3* fornecem dados de coordenadas (transfor-

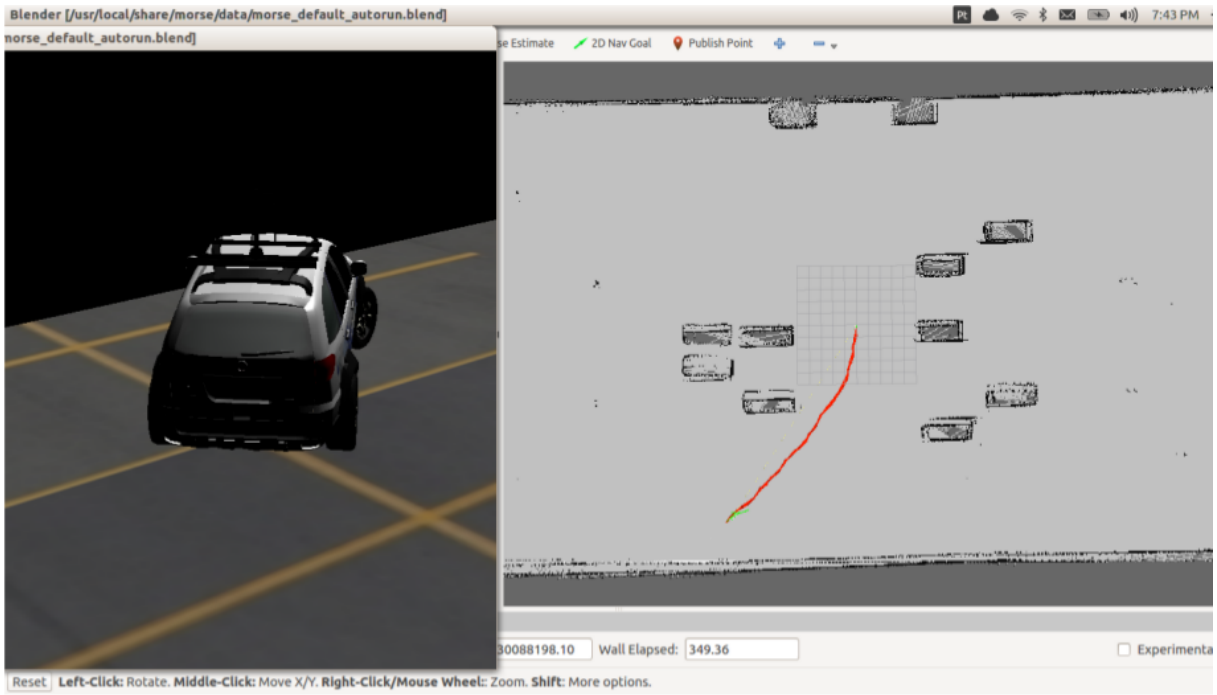

Figura 10 – CaRINA Parking (MATIAS *et al.*, 2015) .

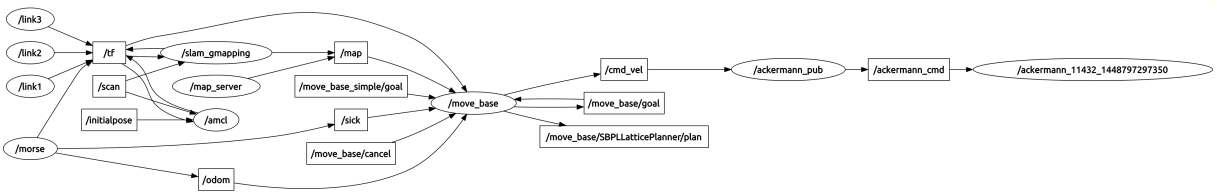

Figura 11 – Grafo gerado pelo ROS para a aplicação do Carina Parking.

mações) para o tópico *tf*. O tópico *tf* recebe dados de coordenadas, que são utilizados durante as transformações de translação e rotação dos diferentes frames. O Nó *slamgmapping* assina o tópico *tf* e com isso publicará dados sobre o mapa e a localização do robô no tópico *map*. Ao iniciar a aplicação o nó *map_server* também publica dados no tópico *map*. O componente *move_base* sobrescreve informações de *map*, *odom*, *sick* e *move_base_goal* necessários para planejar as rotas e publica comandos de velocidade para o robô no tópico *cmd_vel*, que será sobrescrito pelo componente *ackermann_pub*.

A próxima seção apresenta a aplicação dos critérios de teste de programas concorrentes ao CaRINA Parking.

5.3 Aplicação dos Critérios de Teste de Programas Concorrentes

Para aplicação dos critérios de teste de programas concorrentes ao CaRINA Parking, o primeiro passo foi gerar o GFCP adaptado da aplicação. Com base no grafo gerado pelo ROS, apresentado na Figura 11, o robô o CaRINA Parking foi mapeado para um grafo similar ao GFCP, apresentado na Figura 12. No grafo as arestas em vermelho representam serviços do ROS e que não são utilizadas por este estudo. As demais informações do grafo são similares as mapeadas e apresentadas no GFCP da Figura 8 da Seção 4.3.2.1.

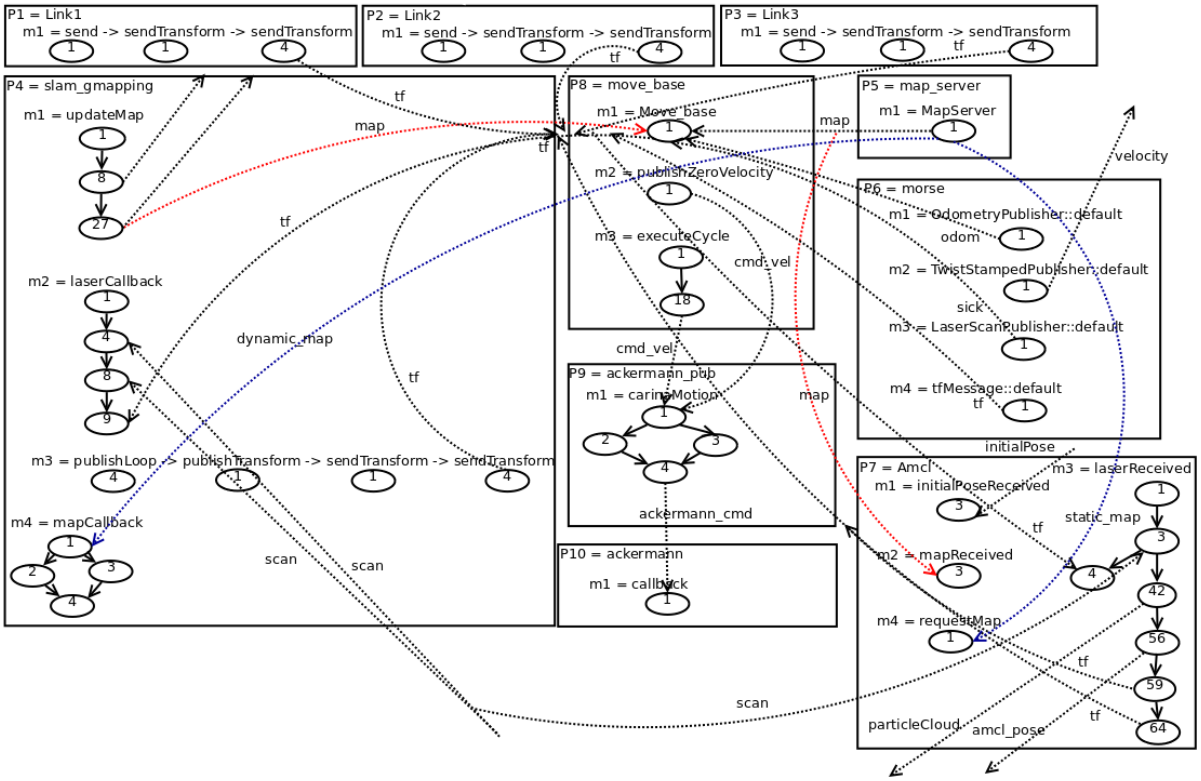

Figura 12 – GFCP para o Carina Parking.

Tabela 6 – Informações obtidas pelo modelo de teste para o robô Carina Parking.

$np = 10$, $Prog = \{p^1, p^3, p^4, p^5, p^6, p^7, p^8, p^9, p^{10}\}$
$p^1 = \{m^1\}$ - $p^2 = \{m^1\}$ $p^3 = \{m^1\}$ - $p^4 = \{m^1, m^2, m^3, m^4\}$ $p^5 = \{m^1\}$ - $p^6 = \{m^1, m^2, m^3, m^4\}$ $p^7 = \{m^1, m^2, m^3, m^4\}$ - $p^8 = \{m^1, m^2, m^3\}$ $p^9 = \{m^1\}$ - $p^{10} = \{m^1\}$
$N^{1,1} = \{n_1^{1,1} ... n_4^{1,1}\}$, $N^{2,1} = \{n_1^{2,1} ... n_4^{2,1}\}$, $N^{3,1} = \{n_1^{3,1} ... n_4^{3,1}\}$, $N^{4,1} = \{n_1^{4,1} ... n_{27}^{4,1}\}$, $N^{4,2} = \{n_1^{4,2} ... n_9^{4,2}\}$, $N^{4,3} = \{n_1^{4,3} ... n_4^{4,3}\}$, $N^{4,4} = \{n_1^{4,4} ... n_4^{4,4}\}$, $N^{5,1} = \{n_1^{5,1}\}$, $N^{6,1} = \{n_1^{6,1}\}$, $N^{6,2} = \{n_1^{6,2}\}$, $N^{6,3} = \{n_1^{6,3}\}$, $N^{6,4} = \{n_1^{6,4}\}$, $N^{7,1} = \{n_1^{7,1} ... n_3^{7,1}\}$, $N^{7,2} = \{n_1^{7,2} ... n_3^{7,2}\}$, $N^{7,3} = \{n_1^{7,3} ... n_6^{7,3}\}$, $N^{7,4} = \{n_1^{7,4}\}$, $N^{8,1} = \{n_1^{8,1}\}$, $N^{8,2} = \{n_1^{8,2}\}$, $N^{8,3} = \{n_1^{8,3} ... n_1^{8,3}\}$, $N^{9,1} = \{n_1^{9,1} ... n_4^{9,1}\}$, $N^{10,1} = \{n_1^{10,1}\}$
$N^1 = N^{1,1}$ $N^2 = N^{2,1}$ $N^3 = N^{3,1}$ $N^4 = N^{4,1} \cup N^{4,2} \cup N^{4,3} \cup N^{4,4}$ $N^5 = N^{5,1}$ $N^6 = N^{6,1} \cup N^{6,2} \cup N^{6,3} \cup N^{6,4}$ $N^7 = N^{7,1} \cup N^{7,2} \cup N^{7,3} \cup N^{7,4}$ $N^8 = N^{8,1} \cup N^{8,2} \cup N^{8,3}$ $\mathrm{N}^9 = N^{9,1}$ $N^{10} = N^{10,1}$ $N = N^1 \cup N^2 \cup N^3 \cup N^4 \cup N^5 \cup N^6 \cup N^7 \cup N^8 \cup N^9 \cup N^{10}$
$E_s = (n_4^{1,1}, n_9^{4,2}), (n_4^{2,1}, n_9^{4,2}), (n_4^{3,1}, n_9^{4,2}), (n_1^{6,1}, n_1^{8,1}), (n_1^{6,3}, n_1^{8,1})(n_1^{6,4}, n_9^{4,2}), (n_{27}^{4,2}, n_1^{8,1})$, $(\mathrm{n}_1^{5,1}, n_1^{8,1}), (n_4^{4,3}, n_4^{7,3}), (n_4^{9,1}, n_1^{10,1}), (n_1^{8,2}, n_1^{9,1}), (n_{18}^{8,3}, n_1^{9,1}), (n_{59}^{7,3}, n_9^{4,2}), (n_{64}^{7,3}, n_9^{4,2})$

A partir do grafo foram obtidas as informações da aplicação. A Tabela 6 contém os conjuntos obtidos a partir das definições no grafo e são utilizados para geração dos requisitos de teste.

5.3.1 Execução dos Testes

A partir das informações geradas foram obtidos os elementos requeridos, apresentados na Tabela 7. Os elementos requeridos foram gerados com base nas definições de programas concorrentes com as adaptações realizadas para sistemas robóticos móveis. Um exemplo da geração dos elementos requeridos para os critérios baseados no fluxo de comunicação é o critério Todas-Arestas-S. Em programas concorrentes os elementos requeridos para este critério são compostos pelos elementos do conjunto E_s, ou seja, todas as arestas de comunicação. Em aplicações robóticas este conceito foi mapeado para todos os tópicos. Um exemplo de um elemento requerido para este critério é $(n_4^{1,1}, n_9^{4,2})$, em que possui um tópico entre os processos P1 e P2, sendo que P1 possui um evento *publish* e P2 possui um evento *subscribe*. Para o critério

Tabela 7 – Alguns elementos requeridos pelos critérios para o robô Carina Parking.

Critério	Elementos requeridos
Todas-Arestas-s	$(n_4^{1,1}, n_9^{4,2}), (n_4^{2,1}, n_9^{4,2}), (n_4^{3,1}, n_9^{4,2}), (n_1^{6,1}, n_1^{8,1}), (n_1^{6,3}, n_1^{8,1})(n_1^{6,4}, n_9^{4,2}),$ $(n_{27}^{4,2}, n_1^{8,1}), (n_1^{5,1}, n_1^{8,1}), (n_4^{4,3}, n_4^{7,3}), (n_4^{9,1}, n_1^{10,1}), (n_1^{8,2}, n_1^{9,1}),$ $(n_{18}^{8,3}, n_1^{9,1}), (n_{59}^{7,3}, n_9^{4,2}), (n_{64}^{7,3}, n_9^{4,2})$
Todos-Nós-r	$(n_9^{4,2})$, $(n_{19}^{5,1})$, $(n_3^{7,2})$, $(n_4^{7,3})$, $(n_1^{8,1})$, $(n_1^{9,1})$, $(n_1^{10,1})$,
Todos-Nós-s	$(n_4^{1,1})$, $(n_4^{2,1})$, $(n_4^{3,1})$, $(n_{27}^{4,1})$, $(n_4^{4,3})$, $(n_9^{4,2}$, $(n_{19}^{5,1})$, $(n_1^{6,1})$, $(n_1^{6,3})$, $(n_1^{6,4})$, $(n_{59}^{7,3})$, $(n_{64}^{7,3})$, $(n_1^{8,2})$, $(n_{19}^{8,3})$, $(n_4^{9,1})$
Todas-Defs/s	$(n_1^{1,1}(n_4^{1,1}, n_9^{4,2}), n_9^{4,2}, tf, tf),$ $(n_1^{2,1}(n_4^{2,1}, n_9^{4,2}), n_9^{4,2}, tf, tf),$ $(n_1^{3,1}(n_4^{3,1}, n_9^{4,2}), n_9^{4,2}, tf, tf),$ $(n_1^{1,1}(n_4^{1,1}, n_9^{4,2}), n_9^{4,2}, tf, tf),$ $(n_1^{8,3}(n_{18}^{8,3}, n_1^{9,1}), n_1^{9,1}, cmd_vel, cmd_vel),$ $(n_1^{6,1}(n_1^{6,1}, n_1^{8,1}), n_1^{8,1}, odom, odom),$ $(n_1^{6,3}(n_1^{6,3}, n_1^{8,1}), n_1^{8,1}, sick, sick)$
Todos-s-Usos	$(n_1^{1,1}(n_4^{1,1}, n_9^{4,2}), tf),$ $(n_1^{2,1}(n_4^{2,1}, n_9^{4,2}), tf),$ $(n_1^{3,1}(n_4^{3,1}, n_9^{4,2}), tf),$ $(n_1^{6,1}(n_6^{1,1}, n_1^{8,1}), odom),$ $(n_1^{6,3}(n_1^{6,3}, n_1^{8,1}), sick),$ $(n_1^{9,1}(n_4^{9,1}, n_1^{10,1}), ackermann_cmd)$
Todos-s-c-Usos	$(n_1^{9,1}(n_4^{9,1}, n_1^{10,1}), n_1^{10,1}, ackermann_cmd, ackermann_cmd)$
Todos-s-p-Usos	$(n_1^{8,2}(n_1^{8,2}, n_1^{9,1}), n_1^{9,1}, cmd_vel, cmd_vel),$ $(n_1^{8,3}(n_{18}^{8,2}, n_1^{9,1}), n_1^{9,1}, cmd_vel, cmd_vel)$

Todos-Nós-R um exemplo da adaptação é apresentado pelo elemento requerido: $(n_9^{4,2})$, em que neste nó há um evento de um *subscribe*. Da mesma maneira ocorre com o critério Todos-Nós-s, que pode ser exemplificado pelo elemento requerido: $(n_4^{1,1})$, onde neste nó há um evento de *publish*.

O mapeamento de uma aplicação real para aplicação dos critérios baseados em fluxo de dados e passagem de mensagens é mais custoso, quando gerado manualmente, pois mapear todas as associações é um grande desafio. Para estes critérios pode-se exemplificar o critério Todos-s-Usos, em que este critério requer que todas as associações s-uso sejam exercitadas. Assim, um requisito de teste para este critério na aplicação robótica pode ser exemplificada por: $(n_1^{1,1}(n_4^{1,1}, n_9^{4,2}), tf)$. Este elemento requerido tem como informações a variável (tf), o nó em que esta variável foi definida $(n_1^{1,1})$, o nó em que houve um evento de *publish* para esta variável $(n_4^{1,1})$, o nó em que esta variável foi recebida por um evento *subscribe* de outro processo $(n_9^{4,2})$ em um caminho livre de definição.

A partir da definição dos elementos requeridos são gerados casos de teste que possam

cobrir estes elementos requeridos. Para gerar casos de teste para a aplicação CaRINA Parking, o primeiro o passo foi entender como ocorre a execução da aplicação e interação entre seus componentes. Com isso, uma entrada para aplicação é um ponto no mapa, que é o destino traçado para a mesma. O resultado esperado para este caso de teste é que o robô pare e estacione no local marcado pelo usuário para simulação.

5.4 Discussões e Análise dos Resultados

Por meio deste estudo foi possível observar que os critérios de teste de programas concorrentes podem ser empregados para o teste de sistemas robóticos móveis. Apesar dos critérios de teste de programas concorrentes serem definidos para o teste de unidade e este trabalho focar no nível de integração, observou-se que a comunicação entre os componentes de um sistema robótico é similar a comunicação realizada entre os processos de um programa concorrente. Para aplicação dos critérios alguns ajustes foram necessários, de maneira a capturar as informações do sistema robótico. Estas adaptações foram realizadas na geração do GFCP.

Uma dificuldade encontrada foi mapear o código e a definição de uso de variáveis de uma aplicação real, pois como é composta por diversos componentes e cada componente pode-se utilizar de diferentes tecnologias para seu desenvolvimento, a geração do GFCP é uma tarefa custosa. Devido à complexidade da aplicação, gerar os elementos requeridos manualmente é um grande desafio, assim como ocorreu na geração do GFCP. O maior desafio é relacionado aos critérios baseados no fluxo de dados e passagem de mensagens, pois extrair todas as informações de definição e uso destas variáveis também é uma tarefa custosa, se realizada manualmente.

5.5 Considerações Finais

Neste capítulo foi apresentado um estudo, em que os critérios de teste de programas concorrentes foram aplicados ao robô CaRINA Parking, o qual é uma aplicação real e desenvolvida para o projeto CaRINA. Este estudo foi importante para este projeto de mestrado, pois foi realizado a aplicação dos critérios de teste estrutural de programas concorrentes para o teste de integração de uma aplicação real de sistemas robóticos móveis, realizando as adaptações necessárias. O objetivo deste estudo é demonstrar a aplicabilidade e adaptabilidade dos critérios de teste definidos para programas concorrentes para testar a integração entre os componentes de um sistemas robótico móvel. Este estudo não contempla a execução dos casos de teste a aplicação, isso se deve muito ao fato do tempo disponível para sua realização e será, portanto, um trabalho futuro deste projeto. Na próximo capítulo serão apresentadas as conclusões deste projeto de mestrado.

CAPÍTULO

6

CONCLUSÕES

6.1 Caracterização da Pesquisa Realizada

O principal objetivo deste projeto de mestrado foi a condução de um estudo sobre a adaptação dos critérios de teste definidos para programas concorrentes para o contexto de robôs móveis. Para isso foi realizado um mapeamento das principais características destes sistemas, levantando suas similaridades e diferenças. Após este passo, foi possível realizar a adaptação dos critérios de teste e apresentar como seria a aplicação destes critérios a sistemas robóticos móveis. Dois estudos de caso foram conduzidos nos quais os critérios propostos para programas concorrentes foram aplicados a sistemas robóticos desenvolvidos em ROS. Como resultados, observou-se os critérios de teste são aplicáveis para testar aspectos de comunicação e sincronização de sistemas robóticos, contribuindo com a qualidade desses sistemas.

Com este estudo foram coletadas informações relevantes que motivaram a definição de critérios de teste, definidos e apresentados em Brito *et al.* (2015). Os novos critérios definidos e que estão sendo explorados em um projeto de doutorado em andamento, foram definidos para tratar aspectos relacionados ao teste de integração e focados no paradigma *publish/subscribe*. Este estudo contribuiu para esta definição e forneceu subsídios para a definição dos novos critérios.

6.2 Contribuições

Destacam-se como principais contribuições deste trabalho:

- Estudo e levantamento de trabalhos relacionados ao teste de integração para sistemas robóticos móveis. Foi conduzido uma pesquisa sobre teste de integração de sistemas embarcados robóticos, em que nesta pesquisa pode-se concluir que não haviam trabalhos

que citam a definição de critérios de teste para o teste de integração de sistemas robóticos móveis.

- Realização de um estudo com as principais características, em que incluem as similaridades e diferenças de sistemas para robôs móveis e programas concorrentes. Este estudo foi de suma importância a este trabalho, pois a partir do mesmo foi possível concluir que as aplicações robóticas e concorrentes possuem similaridades. Com isso, foi possível adaptar os critérios de teste definidos para programas concorrentes para o teste de integração de sistemas robóticos móveis.

- Estudo e aplicação dos critérios de teste de programas concorrentes para o teste de integração de sistemas robóticos móveis.

- Estudo base para definição dos critérios de teste de integração para sistemas robóticos móveis. Após o levantamento das diferenças e similaridades, da aplicação dos critérios de teste de programas concorrentes ao contexto de sistemas robóticos móveis, verificou-se a necessidade de definir novos critérios de teste.

- Estudo e execução dos casos de teste com a utilização das ferramentas Rostest e Gtest. Durante o estudo o Rostest e Gtest foram explorados e utilizados para aplicar os critérios de teste de programas concorrentes à aplicação do Robô Aspirador.

6.3 Desafios, Dificuldades e Limitações

Durante a execução do trabalho alguns desafios, dificuldades e limitações foram encontrados. Algumas dificuldades encontradas foi relacionada às aplicações robóticas e sua execução. Foi necessário um bom tempo para estudo, execução e entendimento de seu funcionamento, pois cada aplicação utiliza tecnologias diferentes, que requerem recursos de hardware avançados e correta configuração do sistema operacional, softwares e ambientes. Um desafio encontrado foi com relação a geração do grafo, pois mapear o código das aplicações, de forma que seja possível gerar o seu grafo é uma tarefa custosa. As aplicações para robôs móveis, desenvolvidas utilizando o ROS, possuem diversos pacotes, diretórios e arquivos de diferentes configurações, formatos e linguagens. Com isso, mapear as chamadas de um método ou componente para outro tornou-se um desafio, pois para isto era necessário compreender cada uma das tecnologias envolvidas.

O mapeamento dos critérios de teste de programas concorrentes para o teste de integração de robôs móveis foi outro desafio deste trabalho, pois além das características presentes nestes sistemas, os critérios de teste definidos para aplicações concorrentes são em nível de unidade e a proposta deste trabalho incluía a adaptação para o nível de integração de componentes de sistemas robóticos móveis. Outro desafio foi definir qual ferramenta utilizar para executar os casos de teste. A princípio a ideia seria desenvolver uma ferramenta para automatizar o processo de execução dos dados de teste. No decorrer da pesquisas e com o direcionamento para aplicações

robóticas que utilizam o ROS, optou-se pela utilização de ferramentas que já são integradas ao mesmo. No entanto, a utilização destas ferramentas é pouco difundida e não se encontram muitos materiais de apoio ou documentação. Com isso, foi despendido uma grande parte do tempo para executar estas ferramentas e analisar seu funcionamento, de forma que fosse possível executar os casos de teste.

Uma das limitações é com relação ao baixo número de trabalhos relacionados. Não foram encontrados muitos trabalhos que descrevem a aplicação de critérios de teste de integração para sistemas robóticos móveis e o uso de frameworks de desenvolvimento para estas aplicações.

6.4 Trabalhos Futuros

Como trabalhos futuros decorrentes dessa dissertação destacam-se: a aplicação dos critérios de teste de programas concorrentes em aplicações robóticas, a definição de ferramentas que permitam a automatização completa do processo de teste de integração e a aplicação dos critérios de teste a outros frameworks de desenvolvimento. Este projeto de mestrado teve como foco aplicações robóticas desenvolvidas no ROS e que utilizam o paradigma de comunicação *publish/subscribe*. Assim, como trabalhos futuros devem ser exploradas a comunicação entre serviços, o qual é outro paradigma de comunicação utilizado entre os componentes do sistema robótico. Como trabalhos futuros sugere-se a execução e exploração de forma experimental para verificar a efetividade da utilização do ROSTEST e Gtest para execução dos casos de teste em apoio a automatização do processo de aplicação dos critérios de teste.

REFERÊNCIAS

BALAKRISHNAN, A.; ANAND, N. Development of an automated testing software for real time systems. In: **International Conference on Industrial and Information Systems (ICIIS)**. [S.l.: s.n.], 2009. p. 193–198. Citado na página 24.

BARBOSA, E. F.; CHAIM M. L. VICENZI, A. M. R.; DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. a. Teste Estrutural. In: DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. (Ed.). **Introdução ao Teste de Software**. [S.l.]: Elsevier, 2007. p. 47 – 76. Citado 2 vezes nas páginas 32 e 33.

BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R.; DELAMARO, M. E.; SOUZA, S. R. S.; JINO, M. Introdução ao teste de software. In: **Minicurso - XIV SBES: Simpósio Brasileiro de Engenharia de Software**. [S.l.: s.n.], 2000. Citado 3 vezes nas páginas 28, 29 e 34.

BEIZER, B. **Software System Testing**. [S.l.]: Van Nostrand Reinhold Company, 1990. Citado na página 23.

BEKEY, G. **Autonomous robots: from biological inspiration to implementation and control**. London, England: MiT Press, 2005. ISBN 0-262-02578-7. Citado na página 34.

BELLI, F.; HOLLMANN, A.; PADBERG, S. Communication sequence graphs for mutation-oriented integration testing. In: **Secure Software Integration and Reliability Improvement. SSIRI. Third IEEE International Conference on**. [S.l.: s.n.], 2009. p. 387–392. Citado 2 vezes nas páginas 50 e 52.

BERGER, A. S. **Embedded Systems Design: An Introduction to Processes, Tools, e Techniques**. [S.l.]: CMP Books, 2002. Citado na página 25.

BRAGA, R. T. V.; BRANCO, K. C.; JR, O. T.; GIMENES, I. M. Evolving Tiriba design towards a product line of small electric-powered UAVs. In: **I Conferência Brasileira de Sistemas Embarcados Críticos**. São Carlos, SP: [s.n.], 2011. p. 67–72. Citado na página 35.

BRANCO, K. R. L. J. C.; PELIZZONI, J. M.; NERIS, L. O.; JR, O. T.; OSóRIO, F. S.; WOLF, D. F. Tiriba a new approach of UAV based on model driven development and multiprocessors. In: **IEEE International Conference on Robotics and Automation (ICRA)**. Shangai: [s.n.], 2011. p. 1–4. Citado na página 35.

BRITO, M. A.; SANTOS., M. P.; SOUZA, S. R.; SOUZA, P. S. Integration testing criteria for mobile robotic systems. In: **SEKE**. [S.l.: s.n.], 2015. p. 182–187. Citado 4 vezes nas páginas 50, 55, 59 e 77.

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. ACEITO. Citado na página 24.

BROEKMAN, B.; NOTENBOOM, E. **Testing Embedded Software**. [S.l.]: Addison-Wesley Professional, 2002. Citado na página 24.

COLLETT, T. H. J.; MACDONALD, B. A. Player 2.0: Toward a practical robot programming framework. In: **in Proc. of the Australasian Conference on Robotics and Automation (ACRA)**. [S.l.: s.n.], 2005. Citado 2 vezes nas páginas 36 e 37.

COWARD, P. A review of software testing. **Information and Software Technology**, v. 30, n. 3, p. 189–198, abr. 1988. Citado na página 23.

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 3 vezes nas páginas 15, 29 e 30.

DELAMARO, M. E.; MALDONADO, J.; JINO, M. **Introdução ao Teste de Software**. 1a. ed. [S.l.]: Rio de Janeiro: Elsevier (Editora Campus), 2007. Citado 4 vezes nas páginas 23, 28, 31 e 34.

DELAMARO, M. E.; MALDONADO, J. C.; MARTHUR, A. P. Interface mutation: An approach for integration testing. **IEEE Trans. Softw. Eng.**, v. 27, n. 3, p. 228–247, 2001. Citado 3 vezes nas páginas 29, 30 e 31.

DEMILLO, R. A. **Software Testing and Evaluation**. [S.l.]: The Benjamin/Cummings Publishing Company, 1987. Citado na página 31.

DEMILLO, R. A.; LIPTON, R. J.; SAYAWARD, F. G. Hints on test data selection: Help for the practicing programmer. **IEEE Computer**, v. 11, n. 4, p. 34–41, abr. 1978. Citado na página 23.

EBERT, C.; SALECKER, J. Guest editors' introduction: Embedded software technologies and trends. **IEEE Softw.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 26, n. 3, p. 14–18, maio 2009. ISSN 0740-7459. Citado na página 25.

ENDO, A. T.; aO, A. S. 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 na página 24.

FABBRI, S. C. P. F.; VINCENZI, A. M. R.; MALDONADO, J. C. Teste funcional. In: DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. (Ed.). **Introdução ao Teste de Software**. [S.l.]: Elsevier, 2007. p. 9 – 24. Citado na página 31.

FERNANDES, A. A. L. **Comunicação Ad Hoc em Equipes de Robôs Móveis Utilizando a Tecnologia ZigBee**. Dissertação (Mestrado) — Departamento de Engenharia Eletrotécnica Faculdade de Ciências e Tecnologia Universidade de Coimbra, 2012. Citado 3 vezes nas páginas 15, 39 e 40.

FERNANDES, L. C.; SOUZA, J. R.; SHINZATO, P. Y.; PESSIN, G.; MENDES, C. C. T.; OSORIO, F. S.; WOLF, D. F. Intelligent robotic car for autonomous navigation: Platform and system architecture. **Brazilian Conference on Critical Embedded Systems**, Los Alamitos, CA, USA, v. 0, p. 12–17, 2012. Citado 2 vezes nas páginas 35 e 38.

FRANCES, C. R. L. Introdução às redes de petri. In: **Universidade Federal do Pará - UFPA**. [S.l.: s.n.], 2000. Citado na página 53.

GERKEY, B. P.; VAUGHAN, R. T.; HOWARD, A. The player/stage project: Tools for multi-robot and distributed sensor systems. In: **In Proceedings of the 11th International Conference on Advanced Robotics**. [S.l.: s.n.], 2003. p. 317–323. Citado na página 37.

GERKEY, K. S. B.; VAUGHAN, R. T. **Player Robot Server**. [S.l.], 2000. Citado na página 37.

GRAMA, A.; ANSHUL, G.; GEORGE, K.; VIPIN, K. **Introduction to Parallel Computing**. [S.l.]: Addison-Wesley, 2003. ISBN ISBN 0-201-64865-2. Citado na página 24.

GRAMA, A.; KARPYS, G.; KUMAR, V.; GUPTA, A. **Introduction to parallel computing**. Second. [S.l.]: Addison Wesley, 2003. Citado na página 57.

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 na página 24.

HERMAN, P. M. A data flow analysis approach to program testing. **Australian Computer Journal**, v. 8, n. 3, nov. 1976. Citado na página 23.

HOSSAIN, M. I.; LEE, W. J. A scalable integration testing approach considering independent usage pattern of global variables. **International Journal of Software Engineering and Its Applications**, p. 195 – 204, 2013. Citado na página 50.

HOWDEN, W. E. Methodology for the generation program test data. **IEEE Transactions on Computing**, v. 24, n. 5, p. 554–560, 1975. Citado na página 23.

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 2 vezes nas páginas 50 e 52.

KANG, S. S.; MAENG, S. W.; KIM, S. W.; PARK, H. S. SITAT: Simulation-based interface testing automation tool for robot software component. In: **Int. Conference on Control Automation and Systems**. [S.l.: s.n.], 2010. p. 1781–1784. Citado 2 vezes nas páginas 50 e 51.

LASKI, J. W.; KOREL, B. A data flow oriented program testing strategy. **IEEE Transactions on Software Engineering**, v. 9, n. 3, maio 1983. Citado na página 23.

LEE, N. H.; KIM, T. H.; CHA, S. D. Construction of global finite state machine for testing task interactions written in message sequence charts. In: **Proceedings of the 14th international conference on Software engineering and knowledge engineering**. New York, NY, USA: ACM, 2002. (SEKE '02), p. 369–376. ISBN 1-58113-556-4. Citado 2 vezes nas páginas 49 e 50.

LIER F., L. I. W. S. Iterative model driven integration checks of component based robotic systems. In: **European Robotics Forum, Robot Engineering Task Force Workshop**. [S.l.: s.n.], 2013. Citado 2 vezes nas páginas 49 e 50.

LILL, R.; SAGLIETTI, F. **Test Coverage Criteria for Autonomous Mobile Systems based on Coloured Petri Nets**. 2012. Citado 2 vezes nas páginas 50 e 52.

____. **Model-based Testing of Cooperating Robotic Systems using Coloured Petri**. 2013. Citado na página 53.

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.; ŚLęZAK, D. (Ed.). [S.l.]: Springer Berlin Heidelberg, 2009, (Lecture Notes in Computer Science, v. 5899). p. 198–207. ISBN 978-3-642-10508-1. Citado 3 vezes nas páginas 15, 50 e 51.

LUCAS, C.; ELBAUM, S.; ROSENBLUM, D. S. Detecting problematic message sequences and frequencies in distributed systems. In: **International Conference on Object Oriented Programming Systems Languages and Applications**. New York, NY, USA: ACM, 2012. p. 915–926. Citado 3 vezes nas páginas 17, 58 e 59.

MALDONADO, J. C. **Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software**. Tese (Doutorado) — DCA/FEE/UNICAMP, jul. 1991. Citado 2 vezes nas páginas 23 e 34.

MATIAS, L. P. N.; SANTOS, T. C.; WOLF, D. F.; SOUZA, J. R. Path planning and autonomous navigation using amcl and ad*. **12th Latin American Robotics Symposium (LARS)**, v. 12, p. 1–6, 2015. Citado 3 vezes nas páginas 15, 71 e 72.

MATTIELLO-FRANCISCO, F.; MARTINS, E.; CAVALLI, A. R.; YANO, E. T. 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. ISSN 0164-1212. Citado na página 50.

MYERS, G. J. **The Art of Software Testing**. 2. ed. New York: John Wiley & Sons, 2004. ISBN 0471469122. Citado 3 vezes nas páginas 23, 28 e 31.

NEVES, V. O. **Automatização do teste estrutural de software de veículos autônomos para apoio ao teste de campo**. Tese (Doutorado) — Instituto de Ciências Matemáticas e de Computação - ICMC-USP, 2015. Citado 2 vezes nas páginas 50 e 53.

NTAFOS, S. C. On required element testing. **IEEE Transactions on Software Engineering**, v. 10, n. 6, nov. 1984. Citado na página 23.

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

PACHECO, P. **An Introduction to Parallel Programming**. [S.l.]: Elsevier, 2011. ISBN 0123742609. Citado na página 24.

PESSIN, G.; OSóRIO, F. S.; WOLF, D. F.; BRASIL, C. R. S. Improving efficiency of a genetic algorithm applied to multi-robot tactic operation. In: **Ibero-American conference on Advances in artificial intelligence**. Berlin, Heidelberg: [s.n.], 2010. p. 50–59. ISBN 3-642-16951-1, 978-3-642-16951-9. Citado na página 35.

PRESSMAN, R. S. **Software Engineering - A Practitioner's Approach**. 5ª. ed. [S.l.]: McGraw-Hill, 2000. Citado 3 vezes nas páginas 32, 33 e 34.

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 2 vezes nas páginas 38 e 39.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. **IEEE Transaction Software Engineering**, v. 11, n. 4, p. 367–375, abr. 1985. Citado 2 vezes nas páginas 23 e 33.

SARMANHO, F. S. **Teste de Programas Concorrentes com Memória Compartilhada**. Dissertação (Mestrado) — Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (ICMC/USP), out. 2009. Citado na página 47.

SARMANHO, F. S.; SOUZA, P. S.; SOUZA, S. R.; aO, A. S. S. Structural testing for semaphore-based multithread programs. In: **8th international conference on Computational Science**. Berlin, Heidelberg: Springer-Verlag, 2008. p. 337–346. Citado 2 vezes nas páginas 24 e 48.

SEO, J.; KI, Y.; CHOI, B.; LA, K. Which spot should i test for effective embedded software testing? In: **Secure System Integration and Reliability Improvement, 2008. SSIRI '08. Second International Conference on**. [S.l.: s.n.], 2008. p. 135–142. Citado na página 50.

SEO, J.; SUNG, A.; CHOI, B.; KANG, S. Automating embedded software testing on an emulated target board. In: **Proceedings of the Second International Workshop on Automation of Software Test**. Washington, DC, USA: IEEE Computer Society, 2007. (AST '07), p. 9–. ISBN 0-7695-2971-2. Citado na página 50.

SHINZATO, P. Y.; OSORIO, F. S.; WOLF, D. F. Visual road recognition based in multiple artificial neural networks. In: **Conferência Brasileira de Sistemas Embarcados Críticos**. São Carlos, SP, Brasil: [s.n.], 2011. p. 26–31. Citado na página 35.

SNIR, M.; OTTO, S.; STEVEN, H.; WALKER, D.; DONGARRA, J. **MPI: The complete reference**. [S.l.], 1996. Citado na página 57.

SOUZA, J. R.; PESSIN, G.; SHINZATO, P. Y.; OSORIO, F. S.; WOLF, D. F. Vision-based autonomous navigation using neural networks and templates in urban environments. In: **Conferência Brasileira de Sistemas Embarcados Críticos**. São Carlos, SP, Brasil: [s.n.], 2011. p. 55–60. Citado na página 35.

SOUZA, P. L.; SAWABE, E. T.; aO, A. S. S.; VERGILIO, S. R.; SOUZA, S. R. S. D. Valipvm - a graphical tool for structural testing of pvm programs. In: **Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface**. Berlin, Heidelberg: Springer-Verlag, 2008. p. 257–264. ISBN 978-3-540-87474-4. Citado na página 48.

SOUZA, P. S.; SOUZA, S. S.; ROCHA, M. G.; PRADO, R. R.; BATISTA, R. N. Data flow testing in concurrent programs with message passing and shared memory paradigms. **Procedia Computer Science**, v. 18, n. 0, p. 149 – 158, 2013. ISSN 1877-0509. Citado na página 24.

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**, p. doi: 10.1002/cpe.2937, 2012. Citado na página 24.

SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, P. S. L.; aO, A. S. S.; BLISCOSQUE, T. G.; LIMA, A. M.; HAUSEN, A. C. Valipar: A testing tool for message-passing parallel programs. In: **International Conference on Software knowledge and Software Engineering (SEKE05)**. Taipei-Taiwan: [s.n.], 2005. p. 386–391. Citado 2 vezes nas páginas 24 e 45.

SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, P. S. L.; aO, A. S. S.; HAUSEN, A. C. Structural testing criteria for message-passing parallel programs. **Concurrency and Computation: Practice and Experience**, v. 20, n. 16, p. 1893 – 1916, mar 2008. Citado na página 24.

SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, S. P. L. Teste de programas concorrentes. In: DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. (Ed.). **Introdução ao Teste de Software**. [S.l.]: Elsevier, 2007. p. 231 – 249. Citado 4 vezes nas páginas 45, 56, 58 e 61.

SUNG, A.; CHOI, B.; WONG, W. E.; 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. ISSN 0950-5849. Citado na página 50.

THRUN, S.; BURGARD, W.; FOX, D. **Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)**. [S.l.]: The MIT Press, 2005. ISBN 0262201623. Citado 2 vezes nas páginas 27 e 34.

TIAN, P.; WANG, J.; LENG, H.; QIANG, K. Construction of distributed embedded software testing environment. **International Conference on Intelligent Human-Machine Systems and Cybernetics**, v. 1, p. 470–473, 2009. Citado na página 25.

TRINDADE, O.; NERIS, L. de O.; BARBOSA, L. C. P.; BRANCO, K. R. L. J. C. A layered approach to design autopilots. In: **IEEE International Conference on Industrial Technology (ICIT)**. [S.l.: s.n.], 2010. p. 1415 –1420. Citado na página 35.

URAL, H.; YANG, B. Structural test selection criterion. **Information Processing Letters**, v. 28, p. 157–163, 1988. Citado na página 23.

VERGILIO, S. R.; SOUZA, S. R. S.; SOUZA, P. S. L. Coverage testing criteria for message-passing parallel programs. In: **LATW2005 - 6th IEEE Latin-American Test Workshop**. Salvador, Ba: [s.n.], 2005. p. 161–166. Citado 2 vezes nas páginas 24 e 46.

WEYUKER, E. J.; WEISS, S. N.; HAMLET, D. Comparison of program testing strategies. In: **Proceedings of the symposium on Testing, analysis, and verification**. New York, NY, USA: ACM, 1991. p. 1–10. ISBN 0-89791-449-X. Citado na página 31.

WOLF, D. F.; OSóRIO, F. S.; SIMõES, E.; JR., O. T. Robótica inteligente: Da simulação às aplicações no mundo real. Editora da PUC Rio, Rio de Janeiro, RJ, Brasil, v. 1, p. 279–330, 2009. Citado 7 vezes nas páginas 24, 27, 34, 35, 36, 37 e 38.

WOLF, W. H. **Computer as Component: principles of embedded computing system design**. 2ª. ed. [S.l.]: Morgan Kaufmann, 2008. Citado na página 25.

XAVIER, J.; PACHECO, M.; CASTRO, D.; RUANO, A. Fast line, arc/circle and leg detection from laser scan data in a player driver. In: **in Proc. of the IEEE Int. Conference on Robotics Automation (ICRA05**. [S.l.: s.n.], 2005. Citado na página 37.

YANG, C.-S.; SOUTER, A. L.; POLLOCK, L. L. All-du-path coverage for parallel programs. In: **International Symposium on Software Testing and Analysis**. [S.l.: s.n.], 1998. p. 153–162. Citado na página 45.

ZHU, H.; HALL, P. A. V.; MAY, J. H. R. Software unit test coverage and adequacy. **ACM Computing Surveys**, v. 29, n. 4, p. 366–427, dez. 1997. Citado na página 31.