Automatização do teste estrutural de software de
veículos autônomos para apoio ao teste de
campo

Vânia de Oliveira Neves

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Vânia de Oliveira Neves

Automatização do teste estrutural de software de veículos autônomos para apoio ao teste de campo

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

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Paulo Cesar Masiero

Coorientador: Prof. Dr. Márcio Eduardo Delamaro

USP – São Carlos
Julho de 2015

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

d499a de Oliveira Neves, Vânia
Automatização do teste estrutural de software de
veículos autônomos para apoio ao teste de campo /
Vânia de Oliveira Neves; orientador Paulo Cesar
Masiero; co-orientador Márcio Eduardo Delamaro. --
São Carlos, 2015.
145 p.

Tese (Doutorado - Programa de Pós-Graduação em
Ciências de Computação e Matemática Computacional) --
Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo, 2015.

1. Teste estrutural de software. 2. Veículos
autônomos. 3. Teste de veículos autônomos. 4. Geração
de dados de teste. 5. Teste baseado em busca. I.
Cesar Masiero, Paulo, orient. II. Eduardo Delamaro,
Márcio, co-orient. III. Título.

Vânia de Oliveira Neves

**Automated structural software testing of autonomous
vehicle to support field testing**

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

Concentration Area: Computer Science and
Computational Mathematics

Advisor: Prof. Dr. Paulo Cesar Masiero
Coadvisor: Prof. Dr. Márcio Eduardo Delamaro

**USP – São Carlos
July 2015**

Agradecimentos

Agradeço à minha família pela formação, amor e carinho dedicados ao longo da minha vida. Ao meu pai, Aparecido, pela torcida, incentivo, apoio incondicional e ajuda nos momentos mais difíceis. À minha mãe, Ivanice, pela dedicação que sempre ofereceu. Ao meu irmão, Fábio, pelo companheirismo e amizade. À minha avó, Magdalena, por ter sido um exemplo de mulher da qual eu me inspiro.

Agradeço ao meu orientador, Paulo Cesar Masiero, pela dedicação, paciência, ética e profissionalismo com que conduziu esta tese, sem moderar esforços. Obrigada por cada vírgula corrigida, pelos conselhos em momentos difíceis, pelas revisões às pressas e/ou de finais de semana e, principalmente, por ter acreditado em mim. Tenho orgulho por ter sido sua aluna de mestrado e doutorado. Ao meu co-orientador, Márcio Eduardo Delamaro, obrigada pelas ideias e contribuições importantes para esta tese. Agradeço também a banca examinadora (Paulo Masiero, Denis Wolf, Marcos Chaim, Thelma Colanzi Lopes e Mariza Bigonha) pelas sugestões e contribuições a esta tese.

Agradeço aos integrantes do LRM/ICMC, Caio Mendes e Patrick Shinzato, e aos professores Denis Wolf e Fernando Osório, pela ajuda com os testes de campo, por fornecerem o programa utilizado nos estudos de caso desta tese e por estarem sempre dispostos a ajudar. Ao Rafael Mantovani, do Biocom/ICMC, pela ajuda com o algoritmo genético.

Agradeço a todos os membros do LabES, onde as pessoas vão além das relações de trabalho e se estabelecem grandes amizades. Nesses mais de sete anos em que tive a honra de fazer parte desse laboratório, a lista de pessoas incríveis que conheci e convivi é enorme e, por isso, não nomearei ninguém aqui pois seria injusta se esquecesse alguém.

Agradeço aos meus amigos de São Carlos e do ICMC, em especial Silvana Morita, Lina Garcés, Valéria Carvalho, Rodolfo Adamshuk, Stevão Andrade, Miguel Yesid, Rafael Oliveira, Rafael Mantovani, Adam Henrique, Merley Conrado, Maria Adelina, Faimison Porto, Silvério Martinez, Marcos Pereira, Jadson Castro, Vinícius Siqueira, Pedro Pereira, Jorge Rebaza, Maria Lucía Castro, Wander Vieira e Nayza Farias. Obrigada pela companhia, conversas e pelos momentos de descontração proporcionados ao longo desses anos. Aos meus grandes amigos André, Beatriz, Edgar, Ivelize, Lilian e Lucas, obrigada por estarem presentes mesmo não estando tão próximos. O meu dia é sempre melhor graças a vocês.

I would like to thank my advisor at Politecnico di Milano, Luciano Baresi, and my colleagues from DEEPSE group: Amir, Adnan, Luca, Mikhail, Federica, Daniel, Leandro, Claudio, Marcello, Santo, Zoya, Mozhdeh, Alessandro, Michele, Mohammad, Manuel,

Danilo, Diego, Andrea e Nicolò. Thank you all for the affection with which you welcomed me and for your contributions during the period for which I was part of that group.

Thanks to the great friends I made in Milan and who have remained despite the time and distance: Adnan Shahzada, Amir Sharifloo, Ana Castellano, Erika Squadrone, Luca Ferrucci and Sarah Collings. Special thanks to Sarah for her corrections and support with English.

Agradeço ao Rafael e aos pais dele, José Roberto e Maria Inês, obrigada pelo incentivo, ajuda e apoio nos primeiros anos deste doutorado.

Agradeço a todos os professores pela minha formação, obrigada pela dedicação que tiveram. Agradeço também aos funcionários da USP São Carlos, em especial aos do ICMC.

Agradeço à FAPESP, CAPES, CNPq e INCT-SEC pelo apoio financeiro.

Veículo autônomo inteligente (ou apenas veículo autônomo – VA) é um tipo de sistema embarcado que integra componentes físicos (hardware) e computacionais (software). Sua principal característica é a capacidade de locomoção e de operação de modo semi ou completamente autônomo. A autonomia cresce com a capacidade de percepção e de deslocamento no ambiente, robustez e capacidade de resolver e executar tarefas lidando com as mais diversas situações (inteligência). Veículos autônomos representam um tópico de pesquisa importante e que tem impacto direto na sociedade. No entanto, à medida que esse campo avança alguns problemas secundários aparecem como, por exemplo, como saber se esses sistemas foram suficientemente testados. Uma das fases do teste de um VA é o teste de campo, em que o veículo é levado para um ambiente pouco controlado e deve executar livremente a missão para a qual foi programado. Ele é geralmente utilizado para garantir que os veículos autônomos mostrem o comportamento desejado, mas nenhuma informação sobre a estrutura do código é utilizada. Pode ocorrer que o veículo (hardware e software) passou no teste de campo, mas trechos importantes do código nunca tenham sido executados. Durante o teste de campo, os dados de entrada são coletados em logs que podem ser posteriormente analisados para avaliar os resultados do teste e para realizar outros tipos de teste *offline*. Esta tese apresenta um conjunto de propostas para apoiar a análise do teste de campo do ponto de vista do teste estrutural. A abordagem é composta por um modelo de classes no contexto do teste de campo, uma ferramenta que implementa esse modelo e um algoritmo genético para geração de dados de teste. Apresenta também heurísticas para reduzir o conjunto de dados contidos em um log sem diminuir substancialmente a cobertura obtida e estratégias de combinação e mutação que são usadas no algoritmo. Estudos de caso foram conduzidos para avaliar as heurísticas e estratégias e são também apresentados e discutidos.

Palavras-chave: Teste estrutural de software; Veículos autônomos; Teste de veículos autônomos; Geração de dados de teste; Teste baseado em busca

Abstract

Intelligent autonomous vehicle (or just autonomous vehicle - AV) is a type of embedded system that integrates physical (hardware) and computational (software) components. Its main feature is the ability to move and operate partially or fully autonomously. Autonomy grows with the ability to perceive and move within the environment, robustness and ability to solve and perform tasks dealing with different situations (intelligence). Autonomous vehicles represent an important research topic that has a direct impact on society. However, as this field progresses some secondary problems arise, such as how to know if these systems have been sufficiently tested. One of the testing phases of an AV is the field testing, where the vehicle is taken to a controlled environment and it should execute the mission for which it was programmed freely. It is generally used to ensure that autonomous vehicles show the intended behavior, but it usually does not take into consideration the code structure. The vehicle (hardware and software) could pass the field testing, but important parts of the code may never have been executed. During the field testing, the input data are collected in logs that can be further analyzed to evaluate the test results and to perform other types of offline tests. This thesis presents a set of proposals to support the analysis of field testing from the point of view of the structural testing. The approach is composed of a class model in the context of the field testing, a tool that implements this model and a genetic algorithm to generate test data. It also shows heuristics to reduce the data set contained in a log without reducing substantially the coverage obtained and combination and mutation strategies that are used in the algorithm. Case studies have been conducted to evaluate the heuristics and strategies, and are also presented and discussed.

Keywords: Structural software testing; Autonomous vehicles; Testing of autonomous vehicles; Test data generation; Search-based testing

Sumário

Abstract	v
Lista de Figuras	xi
Lista de Tabelas	xv
1 Introdução	1
1.1 Contextualização: teste de sistemas embarcados	1
1.2 Motivação	3
1.3 Objetivos	4
1.4 Organização	5
2 Teste estrutural e geração de dados de teste	7
2.1 Considerações iniciais	7
2.2 Definição de teste de software	7
2.3 Programa exemplo	9
2.4 Teste funcional	10
2.5 Teste baseado em defeitos	11
2.6 Teste estrutural	12
2.6.1 Critérios baseados em fluxo de controle	15
2.6.2 Critérios baseados em fluxo de dados	16
2.7 Geração de dados de teste	18
2.7.1 Geração aleatória	19
2.7.2 Geração com execução simbólica	19
2.7.3 Geração com execução dinâmica	19
2.7.4 Geração com algoritmos evolutivos	20
2.7.5 Geração de dados sensíveis a defeitos	22
2.7.5.1 Teste de domínios	22
2.7.5.2 Teste baseado em restrições	23
2.7.5.3 Teste baseado em predicados	23
2.8 Considerações finais	24

3	Teste estrutural de software embarcado com ênfase em veículos autônomos	25
3.1	Considerações iniciais	25
3.2	Mapeamento sistemático sobre teste estrutural em sistemas embarcados . .	26
3.2.1	Fase 1 - Planejamento	26
3.2.2	Fase 2 - Condução	29
3.2.3	Fase 3 - Análise dos resultados	30
3.2.4	Discussão	34
3.3	Teste estrutural em sistemas embarcados	35
3.3.1	Estudos de caso	36
3.3.2	Ferramentas	38
3.3.3	Outros trabalhos	39
3.4	Geração de dados de teste em sistemas embarcados	41
3.4.1	Redução da população inicial	43
3.4.2	Outros trabalhos	44
3.5	Considerações finais	45
4	Veículos autônomos	47
4.1	Considerações iniciais	47
4.2	Veículos autônomos inteligentes	48
4.2.1	Sensores e atuadores	50
4.2.2	Percepção	51
4.2.3	Comportamento, navegação e controle	52
4.2.4	Exemplos de veículos inteligentes	52
4.2.5	Exemplos de arquiteturas de veículos inteligentes	53
4.2.6	Robot Operating System (ROS)	54
4.3	Exemplo do sistema de navegação e desvio de obstáculos do veículo Ca- RINA I	57
4.4	Teste no contexto processo de desenvolvimento de robôs móveis autônomos	58
4.5	Considerações finais	60
5	Ambiente de apoio ao teste de campo de veículos autônomos	63
5.1	Considerações iniciais	63
5.2	Um modelo para apoiar o teste estrutural de veículos autônomos	64
5.3	Implementação da ferramenta desenvolvida	67
5.4	Uso da ferramenta para visualização da cobertura e análise do teste de campo	70
5.5	Considerações finais	74
6	Heurísticas para a redução de dados de teste de softwares controladores de veículos autônomos	77
6.1	Considerações iniciais	77
6.2	Análise de cobertura de testes de campo	78
6.2.1	Análise para os logs particionados em trechos contendo cinco nuvens de pontos	79
6.2.2	Análise para os logs particionados em trechos contendo dez nuvens de pontos	81
6.3	Heurísticas para criação da população inicial: propostas e análises	82

6.3.1	Trechos contendo cinco nuvens de pontos	85
6.3.2	Trechos contendo dez nuvens de pontos	87
6.4	Modelo e ferramenta PATECa	91
6.5	Considerações finais	92
7	Estratégias de combinação e mutação para geração de dados de teste para softwares controladores de veículos autônomos	95
7.1	Considerações iniciais	95
7.2	Geração de novos logs a partir de estratégias de mutação e combinação . .	96
7.2.1	Estratégia de mutação 1: Mudança de Coordenadas - (MC)	98
7.2.2	Estratégia de mutação 2: Rotação Vertical (RV)	99
7.2.3	Estratégia de mutação 3: Rotação Horizontal (RH)	99
7.2.4	Estratégia de mutação 4: Rotação Vertical e Horizontal (RVH) . . .	100
7.2.5	Estratégia de mutação 5: Inserção de Obstáculos (OBS)	101
7.2.6	Estratégia de combinação 6: Média (MED)	103
7.2.7	Estratégia de combinação 7: Troca de Metades Verticais (TMV) . .	103
7.2.8	Estratégia de combinação 8: Troca de Centrais (TC)	104
7.3	Modelo e ferramenta PATECa	104
7.4	Estudo de coberturas obtidas com logs gerados a partir de estratégias de combinação e mutação	106
7.5	Considerações finais	115
8	Geração automática de dados de teste para softwares controladores de veículos autônomos utilizando algoritmos baseados em busca	117
8.1	Considerações iniciais	117
8.2	Algoritmo evolutivo para geração de dados de teste para software controlador de veículos autônomos	118
8.3	Modelo e ferramenta PATECa	122
8.3.1	Dificuldades encontradas	125
8.4	Validação da abordagem proposta	126
8.5	Considerações finais	132
9	Conclusões	133
9.1	Considerações finais	133
9.2	Contribuições desta tese	134
9.3	Trabalhos futuros	135
	Referências	137

Lista de Figuras

2.1	Diagrama de classes do programa exemplo	9
2.2	Código do método <code>calcValorLiquido()</code>	10
2.3	Grafo de fluxo de controle do método <code>calcValorLiquido()</code>	13
2.4	Grafo de fluxo de dados do método <code>calcValorLiquido()</code>	17
3.1	Quantidade de publicações incluídas por critério de inclusão	31
3.2	Estudos primários publicados por ano	31
4.1	Estrutura de um veículo inteligente (retirado de Jung et al. (2005))	49
4.2	Funcionamento básico de um veículo autônomo (retirado de Jo et al. (2014))	50
4.3	Arquitetura de sistema do CaRINA (Fernandes et al., 2014)	54
4.4	Fluxograma da arquitetura de software do Stanley (retirado de Thrun et al. (2006))	55
4.5	Exemplo de nós do ROS	56
4.6	Comunicação entre os nós no ROS	56
4.7	Diagrama de classes	57
4.8	Grafo de chamadas	58
4.9	Processo de desenvolvimento utilizado no desenvolvimento do Boss (Urmson et al., 2008)	59
5.1	Esquema para gerar dados de teste	64
5.2	Um modelo para apoiar uma estratégia de teste estrutural	66
5.3	Um modelo de objetos derivado do metamodelo apresentado na Figura 5.2.	68
5.4	Caso de uso da ferramenta	69
5.5	Arquitetura da ferramenta	69
5.6	Tela de gerenciamento de cenários da PATeCa	71
5.7	Tela de reprodução/simulação da PATeCa	71
5.8	Caminho realizado pelo veículo autônomo durante o teste de campo	72
5.9	Exemplo da reprodução de um teste de campo	73
5.10	Visualização da cobertura de projeto e de classe	73
5.11	Visualização da cobertura de um programa para duas versões diferentes do mesmo programa e dois cenários diferentes.	74
5.12	Visualização da cobertura de um programa para dois logs diferentes.	75

6.1	Cobertura de nós (a) e arestas (b) do método <i>process()</i> com o log2 particionado em tamanho 5	80
6.2	Cobertura de nós (a) e arestas (b) do método <i>calcule()</i> com o log5 particionado em tamanho 5	80
6.3	Cobertura acumulada de nós do método <i>process()</i> para os trechos particionados em tamanho 5 do log2	81
6.4	Cobertura acumulada de arestas do método <i>calcule()</i> para os trechos particionados em tamanho 5 do log5	81
6.5	Cobertura de nós (a) e arestas (b) do método <i>process()</i> com o log3 particionado em tamanho 10	81
6.6	Cobertura de nós (a) e arestas (b) do método <i>calcule()</i> com o log1 particionado em tamanho 10	82
6.7	Cobertura acumulada de nós do método <i>process()</i> para os trechos particionados em tamanho 10 do log3	82
6.8	Cobertura acumulada de arestas do método <i>calcule()</i> para os trechos particionados em tamanho 5 do log1	82
6.9	Cobertura média obtida com as sete heurísticas propostas (a) e média do tamanho dos conjuntos de trechos (b) para trechos particionados com tamanho 5	85
6.10	Cobertura para o critério todos-nós do método <i>process</i> e logs particionados em trechos de tamanho 5	86
6.11	Cobertura para o critério todas-arestas do método <i>calcule</i> e logs particionados em trechos de tamanho 5	86
6.12	Cobertura média obtida com as sete heurísticas propostas (a) e média do tamanho dos conjuntos de trechos(b) para trechos particionados com tamanho 10	88
6.13	Cobertura para o critério todos-nós do método <i>process</i> e logs particionados em trechos de tamanho 10	88
6.14	Cobertura para o critério todas-arestas do método <i>calcule</i> e logs particionados em trechos de tamanho 10	89
6.15	Média das coberturas para o critério todas-arestas do método <i>process</i> e trechos de tamanhos 5 e 10	89
6.16	Média das coberturas para o critério todas-arestas do método <i>calcule</i> e trechos de tamanhos 5 e 10	90
6.17	Média do número de trechos por heurística (a) e média do número de nuvens de pontos por heurística (b)	90
6.18	Modelo atualizado da ferramenta PATeCa	91
6.19	Tela para particionamento em trechos	92
6.20	Tela para geração de logs a partir de heurísticas	92
7.1	Esquema de geração de novos logs que considera apenas as nuvens de pontos geradas	97
7.2	Esquema de geração de novos logs que considera as nuvens de pontos geradas e as nuvens de pontos originais	97
7.3	Exemplo de rotação vertical	99
7.4	Exemplo de rotação horizontal	100

7.5	Exemplo de rotação vertical e horizontal	100
7.6	Exemplo de inserção de um obstáculo que ultrapasse a borda da matriz . .	102
7.7	Ilustração da geração de nuvens de pontos para a estratégia de inserção de obstáculos	102
7.8	Esquema de combinação de pares de nuvens de pontos, incluindo as nuvens de pontos originais	103
7.9	Esquema de combinação de pares de nuvens de pontos, sem incluir as nu- vens de pontos originais	103
7.10	Esquema de combinação de trocas centrais, incluindo as nuvens de pontos originais	104
7.11	Esquema de combinação de trocas centrais, sem incluir as nuvens de pontos originais	104
7.12	Modelo atualizado da PATeCa	105
7.13	Tela de geração de logs a partir de estratégias de combinação e mutação . .	105
7.14	Diagrama de sequência para geração de logs	106
7.15	Gráfico com as médias das coberturas obtidas para a heurística H2	110
7.16	Gráfico com as médias das coberturas obtidas para a heurística H2 do método <i>process()</i> para os critérios a) todos-nós e b) todas-arestas	110
7.17	Gráfico com as médias das coberturas obtidas para a heurística H2 do método <i>calcule()</i> para os critérios a) todos-nós e b) todas-arestas	111
7.18	Gráfico com as médias das coberturas obtidas para a heurística H6	111
7.19	Gráfico com as médias das coberturas obtidas para a heurística H6 do método <i>process()</i> para os critérios a) todos-nós e b) todas-arestas	112
7.20	Gráfico com as médias das coberturas obtidas para a heurística H6 do método <i>calcule()</i> para os critérios a) todos-nós e b) todas-arestas	112
7.21	Gráfico com as coberturas por log e heurística/estratégia do método <i>process()</i> para os critérios a) todos-nós e b) todas-arestas	113
7.22	Gráfico com as coberturas por log e heurística/estratégia do método <i>calcule()</i> para os critérios a) todos-nós e b) todas-arestas	114
8.1	Modelo completo	124
8.2	Tela de geração automática da PATeCa	124
8.3	Tela de gerenciamento dos logs	125
8.4	a) Comparação dos grafos de cobertura de dois logs gerados e b) grafo agregado resultante	125
8.5	Gráfico com as médias das coberturas obtidas para o critério todos-nós do método <i>process</i>	130
8.6	Gráfico com as médias das coberturas obtidas para o critério todas-arestas do método <i>process</i>	131
8.7	Gráfico com as médias das coberturas obtidas para o critério todos-nós do método <i>calcule</i>	131
8.8	Gráfico com as médias das coberturas obtidas para o critério todas-arestas do método <i>calcule</i>	131

Lista de Tabelas

2.1	Cálculo da comissão	10
2.2	Conjunto de casos de teste que satisfazem o critério todos-nós	15
2.3	Conjunto de casos de teste que satisfazem o critério todas-arestas	16
3.1	Quantidade de trabalhos por categoria e fonte	29
3.2	Quantidade de publicações considerando as repetidas	29
3.3	Publicações incluídas	30
3.4	Ferramentas para teste estrutural em sistemas embarcados	39
3.5	Resumo dos principais trabalhos referentes ao teste estrutural de software em sistemas embarcados	46
3.6	Resumo dos principais trabalhos referentes à geração de dados de teste em sistemas embarcados	46
5.1	Esforço de implementação da ferramenta	70
6.1	Dados dos logs e das coberturas dos cinco testes de campo	79
7.1	Cobertura obtida para o critério todos-nós do método <i>process()</i> e heurística H2	108
7.2	Cobertura obtida para o critério todos-nós do método <i>process()</i> e heurística H6	108
7.3	Cobertura obtida para o critério todas-arestas do método <i>process()</i> e heurística H2	108
7.4	Cobertura obtida para o critério todas-arestas do método <i>process()</i> e heurística H6	108
7.5	Cobertura obtida para o critério todos-nós do método <i>calcule()</i> e heurística H2	109
7.6	Cobertura obtida para o critério todos-nós do método <i>calcule()</i> e heurística H6	109
7.7	Cobertura obtida para o critério todas-arestas do método <i>calcule()</i> e heurística H2	109
7.8	Cobertura obtida para o critério todas-arestas do método <i>calcule()</i> e heurística H6	109

7.9	Estratégias que obtiveram melhores resultados para cada heurística e para cada método	115
7.10	Número de vezes em que cada estratégia de mutação foi melhor para cada heurística e para cada método	115
8.1	Variáveis de entrada e tempo de cada execução	128
8.2	Cobertura obtida para o critério todos-nós do método <i>process</i>	129
8.3	Cobertura obtida para o critério todas-arestas do método <i>process</i>	129
8.4	Cobertura obtida para o critério todos-nós do método <i>calcule</i>	130
8.5	Cobertura obtida para o critério todas-arestas do método <i>calcule</i>	130

Lista de abreviaturas e siglas

AG Algoritmo Genético

ATEMES *Automatic Testing Environment for Multi-core Embedded Software*

BCET *Best Case Execution Time*

BOR *Boolean Operator Testing*

BRO *Boolean and Relational Operator Testing*

CACC *Correlated Active Clause Coverage*

CPU *Central Processing Unit*

CRUD *Create, Read, Update and Delete*

DARPA *Defense Advanced Research Projects Agency*

ERTT *Evolutionary Real-Time Testing*

ETF *EvoTest Framework*

FBD *Function Block Diagram*

GCC *GNU Compiler Collection*

GFC Grafo de Fluxo de Controle

GPS *Global Positioning System*

IMU *Inertial Measurement Unit*

INCT-SEC Instituto Nacional de Ciência e Tecnologia em Sistemas Embarcados Críticos

LIDAR *Light Detection And Ranging*

MC Mudança de Coordenadas

MC/DC *Modified Condition/Decision Coverage*

MED Média

MFT *Manual Functional Testing*

OBS Inserção de Obstáculos

PATeCa Pós-Análise de Teste de Campo

PCL *Programmable Logic Controllers*

RANSAC *RANdon SAmple Consensus*

RH Rotação Horizontal

RMA Robô Móvel Autônomo

ROS *Robotic Operating System*

RV Rotação Vertical

RVH Rotação Vertical e Horizontal

SGBD Sistema de Gerenciamento de Banco de Dados

SNB *Super Nested Block*

SRM Sistema Robótico Móvel

TC Troca de Centrais

TMV Troca de Metades Verticais

TSE *Test Set Editor*

UML *Unified Modeling Language*

VA Veículo Autônomo

VFH *Vector Field Histogram*

VV&T Verificação, Validação e Teste

WCET *Worst Case Execution Time*

XML *eXtensible Markup Language*

Introdução

1.1 Contextualização: teste de sistemas embarcados

Sistemas embarcados estão presentes na vida das pessoas de tal forma que é difícil imaginar o dia a dia sem eles. Eles representam uma ampla gama de sistemas que podem ser encontrados em aplicações pessoais, educacionais, da área de saúde, domésticas e industriais; podem ser relativamente simples como controladores de aparelhos eletrodomésticos e segurança residencial ou então complexos e mais robustos como os sistemas de telecomunicações, aviônicos, de suporte à saúde e de apoio à locomoção humana. De maneira geral, um sistema embarcado pode ser considerado como um sistema computacional com propósitos específicos e que deve funcionar de forma autônoma (Jung et al., 2005). Entre suas principais características, um sistema embarcado é composto por um sistema de computação (hardware) projetado e construído especificamente para a aplicação e por um sistema de software que implementa os requisitos funcionais da aplicação (Ebert e Salecker, 2009). O software oferece mais flexibilidade, pois pode ser mais facilmente alterado e o hardware fornece funções mais especializadas e tem uma importância muito grande em sistemas que apresentam algum aspecto crítico, como os relativos ao tempo de resposta.

A complexidade do software embarcado é geralmente maior que outros tipos de software, como sistemas de informação, pois têm características específicas, como exigência de respostas em tempo real e limitações de interface. A maioria dos sistemas embarcados deve ser extremamente confiável porque falhas podem causar danos ao ser humano. E

também porque falhas em campo podem ser muito difíceis ou caras para corrigir como, por exemplo, se o sistema ficar inacessível. Segundo Tian et al. (2009), uma tendência dos sistemas embarcados é que cada vez mais deixarão de ser definidos pelo hardware e passarão a ser projetados para fazer qualquer função que satisfaça objetivos múltiplos e mutáveis. Assim, os sistemas embarcados envolvem muitos desafios que transcendem os requisitos tradicionais das aplicações. Entre esses desafios, está o teste do software embarcado.

Um exemplo de sistema embarcado são os robôs móveis, cuja característica principal é a capacidade de locomoção e de operação de modo semi ou completamente autônomo (Wolf et al., 2009). Atualmente, têm como grandes desafios o desenvolvimento de controle inteligente, que permitam executar tarefas complexas e, principalmente, atuar de modo autônomo, ou seja, sem um constante monitoramento e intervenção humana. Um robô móvel é um sistema embarcado crítico, uma vez que pode causar danos tanto a pessoas como a objetos que o cercam. Dessa maneira, seu desenvolvimento deve envolver um cuidadoso projeto tanto de hardware como de software.

Como consequência, foram criadas diferentes técnicas de teste para sistemas embarcados, algumas focadas no hardware e outras focadas na combinação hardware/software. Além disso, em alguns casos, a combinação entre o hardware e o software é tão próxima de forma que eles devem ser testados em conjunto (Costa e Monteiro, 2013). O teste do software pode usar técnicas funcionais, o que é mais comum, ou técnicas estruturais, baseadas no código. Existem várias pesquisas relacionadas ao teste de sistemas embarcados que buscam aumentar a cobertura do código sem ter como alvo um tipo de falha específico (Costa e Monteiro, 2013). A simulação, ou execução *offline* é uma técnica também muito usada por seu custo inferior e pelas dificuldades para testar em situação real.

Um sistema robótico móvel (SRM ou MRS em inglês) é um tipo de sistema embarcado que tem como objetivo permitir que o dispositivo se mova e faça diversas manobras, como ultrapassar outro veículo, estacionar e obedecer a regras de trânsito sem a presença de um condutor, ou seja, autonomamente, sem a presença de um condutor. Componentes típicos de um SRM são a capacidade de percepção, com sensores aptos a escrutinar e entender o ambiente em que o veículo opera, a capacidade de produzir movimento adequado atuando sobre os dispositivos do veículo, planejamento inteligente para que as ações atômicas atinjam o objetivo geral do veículo e robustez para operar em ambientes muitas vezes hostis. O contexto deste trabalho envolve o teste de SRM, com foco principalmente no software que controla o componente inteligente do veículo, assumindo que os softwares para controle de sensores, atuadores e outras funções básicas do sistema não são tratados individualmente, mas apenas no contexto dos resultados obtidos pelo sistema. Sendo assim, o foco principal do trabalho é no teste de integração e no de sistema. Dependendo

do tipo do robô ou de algumas de suas características, o SRM é chamado de veículo autônomo inteligente, ou simplesmente veículo autônomo (VA).

De um modo geral, após os testes de componentes de hardware e de integração dos componentes, há uma fase de teste baseada em simulação que pode usar dados do ambiente de simulação ou então dados de teste reais (Rauskolb et al., 2008; Thrun et al., 2006; Trepagnier et al., 2006; Urmson et al., 2008). Um tipo de teste comum nesse ambiente é chamado de “teste de campo”, em que um certo cenário de teste é preparado (também chamado de missão a ser cumprida) como, por exemplo: atingir um ponto final em uma área trafegável, a partir de um ponto inicial e evitando obstáculos. Um teste como esse é avaliado principalmente de forma funcional, verificando-se se o objetivo foi atingido ou não. Mesmo que o objetivo tenha sido atingido, é importante também avaliar estruturalmente o código e quais trechos foram ou não executados no teste. Durante um teste de campo, podem ser coletados dados de entrada e armazenados em um arquivo de registro (*log*). É importante notar que esse tipo de teste tem algumas características especiais, pois não usa um conjunto de dados de entradas previamente preparado pelo testador, mas sim dados do ambiente recebidos durante a execução do programa e eles quase sempre são diferentes em cada teste efetuado, trata de estruturas de dados complexas e é dificilmente repetido da mesma forma.

1.2 Motivação

O grupo de Robótica Móvel do ICMC/USP desenvolve pesquisas em diversas áreas relacionadas à robótica. Entre elas destacam-se: visão computacional, sistemas inteligentes, computação evolutiva, aprendizado de máquina, sistemas computacionais reconfiguráveis, controle de veículos aéreos e terrestres autoguiados e automação. O grupo conta com robôs e sensores avançados para o desenvolvimento e teste dos sistemas desenvolvidos. O grupo de Engenharia de Software do ICMC/USP, por sua vez, vem desenvolvendo várias pesquisas sobre o teste estrutural e, nessa linha, foram feitas propostas e desenvolvidas diversas ferramentas para apoiar essa técnica de teste (Vincenzi et al. (2006); Lemos et al. (2007); Lemos et al. (2009); Neves et al. (2009)).

Tanto o grupo de Engenharia de Software quanto o grupo de Robótica Móvel participaram do Instituto Nacional de Ciência e Tecnologia em Sistemas Embarcados Críticos (INCT-SEC). O INCT-SEC é uma criação do Ministério de Ciência e Tecnologia em parceria com instituições de fomento (CNPq e FAPESP) e constitui uma rede de pesquisa para trabalhos na área de Sistemas Embarcados Críticos, que apoia o desenvolvimento de áreas estratégicas do país, como controle do meio ambiente, segurança e defesa nacional e agricultura. Esta tese está relacionada ao INCT-SEC, cujo objetivo é desenvolver pesqui-

sas para a construção de sistemas embarcados críticos com ênfase em veículos autônomos móveis aéreos e terrestres.

Como sistemas embarcados críticos podem causar danos tanto às pessoas quanto aos objetos em seu ambiente de atuação, eles enfrentam exigências de qualidade elevada. Assim, é de fundamental importância que esses sistemas atendam seus requisitos. Se o software embarcado tem qualidade insuficiente, prejuízos graves poderão ocorrer, com ferimentos, morte ou até catástrofes (Ebert e Salecker, 2009). Com isso, é de suma importância que esses sistemas tenham sido suficientemente testados. Assim, surgiu a motivação para a investigação do teste estrutural em sistemas embarcados, em especial do software de controle de veículos autônomos.

Técnicas de teste são complementares e, portanto, além da técnica funcional, ser capaz de aplicar a técnica estrutural em sistemas embarcados é importante. Em particular, garante que trechos críticos do código tenham sido exercitados. É também ponto essencial quando se busca certificação em domínios críticos, como o aeroespacial, e deve obedecer a normas e padrões como a RTCA/DO - 178B (RTCA.org, 2015) e submeter o sistema a processos de certificação.

1.3 Objetivos

O objetivo geral desta tese é apresentar uma abordagem de teste para veículos autônomos composta por um modelo, uma ferramenta, heurísticas, estratégias e algoritmos para apoiar o uso de teste estrutural de software para sistemas embarcados no domínio de veículos autônomos. Mais especificamente, para apoiar a fase do processo de desenvolvimento de veículos autônomos que envolve a análise dos resultados obtidos em testes de campo com o veículo. Para isso, são investigados os problemas relacionados ao teste estrutural nesse domínio e são apresentados estudos práticos realizados para entender as características do teste e as necessidades dos testadores nesse contexto.

Para alcançar o objetivo geral, a autora desta tese trabalhou em colaboração com grupos que desenvolvem veículos autônomos e teve acesso a programas e ambientes de desenvolvimento desse domínio e se dedicou ao desenvolvimento de uma ferramenta de apoio à análise dos resultados de teste de campo com o objetivo de ampliar a cobertura estrutural do teste. Nesse contexto foram investigadas formas de visualização dos resultados do teste de campo, redução dos logs de entrada capturados durante o teste de campo, um algoritmo genético para gerar dados de teste adicionais e estratégias de combinação e mutação nesse domínio.

1.4 Organização

No Capítulo 2 apresenta-se uma revisão dos principais conceitos envolvendo teste estrutural de software e geração de dados de teste com o objetivo de facilitar a compreensão da tese.

No Capítulo 3 é apresentado um mapeamento sistemático realizado com o objetivo de identificar as pesquisas existentes e os problemas de pesquisa de teste estrutural no domínio de sistemas embarcados. São também definidas e discutidas as características de sistemas embarcados, com enfoque em veículos autônomos.

O Capítulo 4 contém uma análise das características do software e do processo de desenvolvimento de veículos autônomos com ênfase em teste. Uma descrição detalhada do teste de campo é apresentada, com foco nas análises que podem ser feitas após a realização do teste.

No Capítulo 5 é apresentado um modelo de classes que representa os principais conceitos envolvidos no teste de campo e na sua análise posterior. Esse modelo é a base para uma ferramenta, cujo nome é PATeCa (Pós-Análise de Teste de Campo) que foi desenvolvida e é apresentada parcialmente neste capítulo. As funcionalidades que permitem a visualização dos resultados do teste de campo em termos estruturais são apresentadas e discutidas.

No Capítulo 6 é proposto um algoritmo para reduzir os logs de nuvens de pontos coletados durante o teste de campo de tal forma que um pequeno subconjunto possa produzir uma cobertura semelhante à de todo o log, para os critérios todos-nós e todas-arestas. São também apresentadas análises de heurísticas para criar o conjunto reduzido. Mostra-se também como esse algoritmo foi implementado na ferramenta proposta.

No Capítulo 7 são definidas várias estratégias de mutação e de combinação de nuvens de pontos que foram implementadas na ferramenta proposta. Apresenta-se também um estudo de caso realizado com o objetivo de analisar e comparar quais produzem melhor cobertura segundo os critérios propostos.

No Capítulo 8 é apresentado e discutido um algoritmo genético para gerar dados de teste para o domínio de veículos autônomos. O algoritmo usa uma população inicial que pode ser criada da forma sugerida no Capítulo 5 e pode usar diferentes subconjuntos de estratégias de combinação e mutação dentre aquelas definidas no Capítulo 7. Um amplo estudo para a validação do algoritmo e para avaliar quais são as melhores estratégias de combinação e de mutação é também apresentado.

O Capítulo 9 apresenta as considerações finais da tese, as suas principais contribuições e pesquisas que podem ser conduzidas em continuação.

Teste estrutural e geração de dados de teste

2.1 Considerações iniciais

Neste capítulo são apresentados os principais conceitos de teste de software relevantes para esta tese. Ele está organizado da seguinte forma: na Seção 2.2 apresenta-se uma visão geral de teste de software; na Seção 2.3 apresenta-se um exemplo que será utilizado ao longo deste capítulo; as Seções 2.4 e 2.5 apresentam, resumidamente, as técnicas de teste funcional e baseada em defeitos, respectivamente; as Seções 2.6 e 2.7 tratam, respectivamente, da técnica de teste estrutural e da geração de dados de teste.

2.2 Definição de teste de software

Muitos programadores definem a atividade de teste como uma atividade que deve ser realizada para mostrar que erros não estão presentes ou, ainda, acredita que a atividade de teste tem o propósito de demonstrar que o programa realiza suas funções corretamente. Essas são ideias equivocadas, uma vez que o processo de construção de um software depende da habilidade das pessoas que o constroem, bem como da competência delas para interpretar o processo e executá-lo. Dessa forma, é indiscutível que os erros humanos sempre acabam surgindo, mesmo com a utilização das melhores ferramentas, técnicas e

CAPÍTULO 2. TESTE ESTRUTURAL E GERAÇÃO DE DADOS DE TESTE

métodos. Desse ponto de vista, Myers (2004) define o teste de software como “o processo de execução de um programa com a intenção de encontrar erros.”

Com o intuito de descobrir erros antes da utilização do software e garantir que ele está em conformidade com o que foi especificado, várias atividades têm sido utilizadas (que estão dentro do contexto da grande área denominada Garantia de Qualidade de Software), chamadas de “Verificação, Validação e Teste” (ou “VV&T”), em que Verificação refere-se ao conjunto de atividades que garantem que uma aplicação implementa corretamente uma função específica; Validação refere-se ao conjunto de atividades que garantam que a aplicação desenvolvida corresponda aos requisitos (Rocha et al., 2001) e o Teste refere-se ao processo de execução de um programa com a finalidade de encontrar erros. As atividades de VV&T podem e devem ser aplicadas durante todo o processo de desenvolvimento do software, não se restringindo apenas ao produto final.

As atividades de VV&T podem ser divididas em estáticas e dinâmicas. As atividades estáticas não requerem a execução ou mesmo a existência de um programa ou modelo executável para serem conduzidas. A inspeção de software, por exemplo, é uma atividade estática. As atividades dinâmicas se baseiam na execução de um programa ou de um modelo. O teste, por exemplo, é uma atividade dinâmica.

Para testar um software, executa-se um programa ou um modelo com a utilização de algumas entradas em particular; em seguida, verifica-se se o resultado obtido corresponde ao esperado. Caso isso não ocorra, a existência de um defeito foi revelada.

Alguns dos termos utilizados no contexto de teste de software dão a impressão que têm o mesmo significado e muitas vezes são utilizados assim, mas foram padronizados pela IEEE (IEEE, 1990) com definições diferentes. São eles:

- Defeito (*fault*): passo, processo ou definição de dados incorretos;
- Engano (*mistake*): ação humana que produz um resultado incorreto;
- Erro (*error*): ocasionado pela existência de um defeito durante a execução de um programa, que se caracteriza por um estado inconsistente ou inesperado; e
- Falha (*failure*): produção de uma saída incorreta em relação à especificação. Um erro pode levar a uma falha.

Não se pode afirmar que um software não tem defeitos com base na condução de uma atividade de teste, mas é possível mostrar a presença deles, caso existam. O que se pode obter é um grau maior de confiança no programa, caso o teste tenha sido realizado de forma criteriosa e embasada tecnicamente.

Um dos desafios da atividade de teste é que é sempre impossível testar um programa de forma a encontrar todos os seus erros. Para evitar esse problema, foram estabelecidas algumas técnicas de teste, entre as quais o teste funcional, o teste estrutural e o teste baseado em defeitos, apresentados nas próximas seções. Antes da apresentação dessas técnicas, é apresentada a especificação de um programa utilizado nesta tese para exemplificar alguns dos critérios definidos.

2.3 Programa exemplo

O programa P deverá controlar a venda de produtos e ser capaz de calcular a comissão dos vendedores sobre a venda e o valor líquido de cada venda, que é o valor total descontado da comissão do vendedor. A Figura 2.1 apresenta o diagrama de classes deste programa.

Figura 2.1: Diagrama de classes do programa exemplo

Ao realizar uma venda, o vendedor e os itens devem ser informados. O valor mínimo de cada item é definido pelo atributo **preco** da classe **Produto**. Cada item pode ser vendido por um preço maior, mas nunca por um preço menor. A comissão do vendedor deverá ser calculada de três diferentes formas, dependendo do lucro obtido com as vendas do mês anterior e da quantidade de itens distintos vendidos, conforme apresentado na Tabela 2.1:

A Figura 2.2 apresenta o código do método `calcValorLiquido()` da Classe **Venda**:

Como esse programa foi construído com o intuito de exemplificar os critérios de teste, algumas construções podem não ser as mais adequadas.

Tabela 2.1: Cálculo da comissão

Lucro obtido pelo vendedor	Qtde de itens vendidos	Perc. de comissão
[0,00;1000,00]	[0,10]	10% sobre o valor total da venda
[0,00;1000,00]	$(10, x] \mid x \in \mathbb{N} \text{ e } x > 10$	15% sobre o valor total da venda
(1000,00; 5000,00]	$x \mid x \in \mathbb{N}$	15% sobre o valor total da venda
$(5000,00, y] \mid y \in \mathbb{R}^+ \text{ e } y > 5000,00$	[0,10]	10% sobre o valor do preço mínimo de cada item mais 50% sobre a diferença
$(5000,00, y] \mid y \in \mathbb{R}^+ \text{ e } y > 5000,00$	$(10, x] \mid x \in \mathbb{N} \text{ e } x > 10$	15% sobre o valor do preço mínimo de cada item mais 50% sobre a diferença

```

1      public double calcValorLiquido(double l, int q){
2      /* 1 */    double comissao = 0;
3      /* 1 */    double lucroAnterior = l;
4      /* 1 */    int qtde = q;
5      /* 1 */    double percFaixa1 = 0.10;
6      /* 1 */    double percFaixa2 = 0.15;
7      /* 1 */    double percFaixa3 = 0.5;
8      /* 1 */    if (lucroAnterior <= 1000){
9      /* 2 */        if (qtde <= 10)
10     /* 3 */            comissao = this.getValorTotal()*percFaixa1;
11     /* 4 */        else
12     /* 4 */            comissao = this.getValorTotal()*percFaixa2;
13     /* 4 */        }
14     /* 5 */    else
15     /* 5 */        if ((lucroAnterior > 1000) && (lucroAnterior <= 5000)){
16     /* 6 */            comissao = this.getValorTotal()*percFaixa2;
17     /* 6 */        }
18     /* 7 */    else {
19     /* 7 */        int qtdeAux = qtde;
20     /* 8 */        while (qtdeAux > 0) {
21     /* 9 */            Item item = this.retornaItem(qtdeAux);
22     /* 9 */            double valorItem = item.getProduto().getPreco()*item.getQuantidade();
23     /* 9 */            if (qtde <= 10) {
24     /* 10 */                comissao = comissao + valorItem*percFaixa1 +
25     /* 10 */                    (item.getValorTotal() - valorItem)*percFaixa3;
26     /* 10 */            }
27     /* 11 */            else{
28     /* 11 */                comissao = comissao + valorItem*percFaixa2 +
29     /* 11 */                    (item.getValorTotal() - valorItem)*percFaixa3;
30     /* 11 */            }
31     /* 12 */            qtdeAux--;
32     /* 12 */        }
33     /* 12 */    }
34     /* 13 */    return (this.ValorTotal - comissao);
35     /* 13 */ }

```

Figura 2.2: Código do método calcValorLiquido()

2.4 Teste funcional

A técnica de teste funcional é também conhecida como teste caixa-preta, uma vez que ela é indiferente à estrutura e ao comportamento interno do programa. Por esse motivo,

o comportamento de um programa em teste só pode ser determinado a partir de suas entradas e saídas relacionadas.

Uma das maneiras de testar um software utilizando essa técnica é submeter o programa a todas as entradas possíveis. Essa abordagem, chamada de teste exaustivo, é inviável, pois o domínio de entrada pode ser infinito ou muito extenso, levando a atividade de teste a consumir muito tempo. Por isso, utilizam-se critérios de teste, entre os quais, os mais conhecidos da técnica funcional são: particionamento em classes de equivalência, análise do valor limite e grafo causa-efeito.

Os critérios de teste funcional baseiam-se na especificação do produto a ser testado. Dessa forma, é fácil perceber que a qualidade desses critérios depende da existência de uma boa especificação de requisitos. Em contrapartida, como não levam em conta detalhes de implementação, esses critérios podem ser aplicados em qualquer fase de teste e em produtos desenvolvidos em qualquer paradigma de programação.

2.5 Teste baseado em defeitos

Nessa técnica, os requisitos de teste são gerados pela utilização de defeitos típicos do processo de implementação de software. Com a utilização dele é possível medir a qualidade de um conjunto de testes, considerando sua efetividade ou sua habilidade em detectar defeitos (Zhu et al., 1997).

O teste de mutação, também conhecido como análise de mutantes, é um dos critérios dessa técnica e tem por objetivo alterar o programa que está sendo testado por diversas vezes para construir uma coleção de programas alternativos, chamados de programas mutantes, como se estivessem introduzindo defeitos no programa original. Dado um conjunto de casos de teste para esse programa, cada mutante é executado sobre cada membro do conjunto. A execução para quando o mutante e o programa produzem respostas diferentes (neste caso, diz-se que o mutante “morreu”); ou quando os casos de teste se esgotam (nesse caso, diz-se que o mutante está “vivo”). As razões para que um mutante permaneça vivo são que ou os dados de teste são inadequados ou que o programa mutante é equivalente ao original, ou seja, ele apresenta o mesmo comportamento que o programa original independente da entrada. Para aplicar esse critério, são necessários os seguintes passos (Delamaro et al., 2007):

1. Geração dos mutantes: em que são construídos uma coleção de programas alternativos que diferem do programa original por uma pequena diferença sintática em um comando;

2. Execução do programa em teste: assim como nas outras técnicas, consiste em executar o programa original usando os casos de teste selecionados e verificar se seu comportamento é o esperado;
3. Execução dos mutantes: cada um dos mutantes é executado usando um conjunto de casos de teste $\mathbf{T}$. Se um mutante $\mathbf{M}$ apresenta resultado diferente do programa original $\mathbf{P}$, isso significa que os casos de teste utilizados são sensíveis e conseguiram expor a diferença entre $\mathbf{P}$ e $\mathbf{M}$. Nesse caso, $\mathbf{M}$ está morto e é descartado. Por outro lado, se $\mathbf{M}$ apresenta comportamento igual a $\mathbf{P}$, diz-se que $\mathbf{M}$ continua vivo. Isso indica uma fraqueza em $\mathbf{T}$, pois não conseguiu distinguir $\mathbf{P}$ de $\mathbf{M}$, ou que $\mathbf{P}$ e $\mathbf{M}$ são equivalentes. Após a execução dos mutantes, pode-se ter uma ideia da adequação dos casos de teste utilizados, por meio do “escore de mutação”. O escore de mutação varia entre 0 e 1 e fornece uma medida objetiva de quanto o conjunto de casos de teste analisado aproxima-se da adequação. Dado o programa $\mathbf{P}$ e o conjunto de casos de teste $\mathbf{T}$, calcula-se o escore de mutação $ms(P, T)$ da seguinte maneira:

$$ms(P, T) = \frac{DM(P, T)}{M(P) - EM(P)}, \text{ em que:}$$

$DM(P, T)$: número de mutantes mortos pelo conjunto de casos de teste $\mathbf{T}$;

$M(P)$: número total de mutantes gerados a partir do programa $\mathbf{P}$; e

$EM(P)$: número de mutantes gerados que são equivalentes a $\mathbf{P}$.

4. Análise dos mutantes: em que primeiro é preciso decidir se o teste continua ou não. Se o escore de mutação for 1 (ou estiver próximo de 1, de acordo com a avaliação do testador), então, pode-se encerrar o teste e considerar $\mathbf{T}$ um bom conjunto de casos de teste para $\mathbf{P}$. Caso se decida continuar o teste, é preciso analisar os mutantes que sobreviveram à execução com os casos de teste disponíveis e decidir se esses mutantes são ou não equivalentes ao programa original.

Embora esse critério tenha sido originalmente desenvolvido para o teste de unidade, é possível adaptá-lo para outras fases do processo de software. Levando isso em consideração, o critério Mutação de Interface tem como objetivo testar as interações com as outras unidades do programa no nível de integração. Para isso, procura, por meio de injeção de defeitos simples, introduzir perturbações somente em objetos relacionados com a interação entre as unidades (Delamaro et al., 2007).

2.6 Teste estrutural

O teste estrutural, também conhecido como teste caixa-branca, utiliza a estrutura lógica do programa para gerar os casos de teste. Essa técnica é tida como complementar à de

teste funcional, uma vez que a natureza dos defeitos encontrados por essas técnicas é diferente.

Para representar a estrutura interna do programa é utilizado um Grafo de Fluxo de Controle (GFC) ou Grafo de Programa, a partir do qual os critérios de teste são estabelecidos. A Figura 2.3 apresenta o grafo de fluxo de controle do método `calcValorLiquido`, apresentado na Figura 2.2. Os critérios de teste estrutural classificam-se em três categorias: com base na complexidade, no fluxo de controle e no fluxo de dados. Cada uma delas é descrita adiante com mais detalhes. Antes disso, são apresentadas algumas definições de conceitos, obtidas de Barbosa et al. (2007), essenciais para o entendimento desses critérios.

Figura 2.3: Grafo de fluxo de controle do método `calcValorLiquido()`

Bloco de comandos: um programa P pode ser decomposto em um conjunto de blocos disjuntos de comandos nos quais 1) a execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco, na ordem dada; e 2) não é possível executar um comando sem que seu antecessor no bloco seja executado imediatamente antes. Dessa forma, todos os comandos de um bloco (com exceção do primeiro) têm um único predecessor e um único sucessor (com exceção do último). Na Figura 2.2, as linhas 2 a 8 representam um bloco de comandos.

Um programa P pode ser representado como um GFC $G = (N, E, s)$, em que:

- N é o conjunto de nós. Cada nó representa um bloco indivisível de comandos;

- E representa o conjunto de arestas direcionadas. As arestas representam o fluxo de controle do programa, ou seja, um possível desvio de um bloco para outro;
- s representa o nó de entrada, ou seja, o nó que corresponde ao bloco da primeira instrução da unidade.

Caminho: é uma sequência finita de nós $(n_1, n_2, \dots, n_k)$, $k \geq 2$, tal que existe uma aresta de n_i para n_{i+1} para $i = 1, 2, \dots, k-1$.

Caminho simples: um caminho em que todos os nós que o compõem são distintos (com exceção possivelmente do primeiro e do último).

Caminho livre de laço: um caminho em que todos os nós, incluindo o primeiro e o último, são distintos.

Caminho completo: um caminho em que o primeiro nó é o de entrada e o último é o nó de saída (ou seja, um nó que não tem sucessores) do grafo G.

Exemplo: O caminho (1,2,4,13) do grafo da Figura 2.3 é um caminho simples, livre de laço e completo.

Definição de variável: um tipo de ocorrência de uma variável em um programa em que um valor é armazenado em uma posição de memória. Na Figura 2.2, o comando `double lucroAnterior = 1;`, contém uma definição da variável `lucroAnterior`.

Uso da variável: segundo e último tipo de ocorrência de uma variável em um programa. Ocorre quando a referência a essa variável não a estiver definindo. Pode ser de dois tipos:

- Uso predicativo (p-uso): afeta diretamente o fluxo de controle do programa. O comando `if (lucroAnterior <= 1000)` na Figura 2.2 (linha 8), corresponde a um uso predicativo da variável `lucroAnterior`.
- Uso computacional (c-uso): qualquer uso que não seja predicativo. Afeta diretamente uma computação realizada ou permite que o resultado de uma definição anterior possa ser observado. O comando `comissao = this.getValorTotal()*percFaixa1;` na Figura 2.2 (linha 10), corresponde a um uso computacional da variável `percFaixa1`;

Caminho livre de definição para uma variável x dos nós i a j é um caminho $(i, n_1, \dots, n_m, j)$, $m \geq 0$, que não contem definição de x nos nós $n_1, \dots, n_m$.

Um nó i tem uma **definição global** de uma variável x se ocorre uma definição de x no nó i e existe um caminho livre de definição i para algum nó ou para algum arco que contém um c-uso ou um p-uso, respectivamente, da variável x .

C-uso global: é um c-uso da variável x em um nó j em que não existe uma definição de x no nó j precedendo esse c-uso.

C-uso local: é um c-uso da variável x em um nó j em que existe uma definição de x no nó j precedendo esse c-uso.

du-caminho em relação à variável x é um caminho $(n_1, n_2, \dots, n_j, n_k)$ em que n_1 tem uma definição global de x e: (1) n_k tem um c-uso de x e $(n_1, n_2, \dots, n_j, n_k)$ é um caminho simples livre de definição com relação a x ; ou (2) (n_j, n_k) tem um p-uso de x e $(n_1, n_2, \dots, n_j, n_k)$ é um caminho livre de definição com relação a x e $n_1, n_2, \dots, n_j$ é um caminho livre de laço.

2.6.1 Critérios baseados em fluxo de controle

Os critérios dessa categoria utilizam características do controle da execução do programa para derivar os requisitos de teste. Os mais conhecidos são:

- **Todos-nós:** exige que a execução do programa passe, ao menos uma vez, em cada vértice do GFC; ou seja, que cada comando do programa seja executado ao menos uma vez. A cobertura desse critério é o mínimo esperado em uma atividade de teste. Considerando o grafo de programa apresentado como exemplo na Figura 2.3, alguns dados de entrada (e seus respectivos nós) necessários para satisfazer esse critério estão listados na Tabela 2.2;
- **Todas-arestas (ou todos-arcos):** requer que cada aresta do grafo, isto é, cada desvio do fluxo de controle do programa, seja exercitada pelo menos uma vez. No grafo de exemplo apresentado na Figura 2.3, os dados de entrada que fazem com que este critério seja satisfeito são listados na Tabela 2.3;
- **Todos-caminhos:** requer que todos os caminhos possíveis do programa sejam executados. Executar todos os caminhos de um programa é uma tarefa impraticável (embora desejável) uma vez que o número de caminhos é muito grande ou até mesmo infinito. Tomando por base o grafo de programa da Figura 2.3, o número de caminhos neste exemplo pode ser infinito devido a existência de um *loop* (por exemplo, os caminhos $(8,9,10,12,8)$ e $(8,9,11,12,8)$).

Tabela 2.2: Conjunto de casos de teste que satisfazem o critério todos-nós

Entrada	Nós
(800,5)	1,2,3,13
(800,11)	1,2,4,13
(1100, 10)	1,5,6,13
(5200, 1)	1,5,7,8,9,10,12,8,13
(5200, 11)	1,5,7,8,9,11,12,8,13

Tabela 2.3: Conjunto de casos de teste que satisfazem o critério todas-arestas

Entrada	Arestas
(800,5)	(1-2),(2-3),(3-13)
(800,11)	(1-2),(2-4),(4-13)
(1100, 10)	(1-5),(5-6),(6-13)
(5200, 1)	(1-5),(5-7),(7-8),(8-9),(9-10),(10-12),(12-8),(8-13)
(6400, 11)	(1-5),(5-7),(7-8),(8-9),(9-11),(11-12),(12-8),(8-13)

2.6.2 Critérios baseados em fluxo de dados

Uma das motivações para a introdução dos critérios baseados em fluxo de dados é que é impraticável executar todos os caminhos de um programa. Além disso, os critérios de fluxo de controle todos-nós e todas-arestas são pouco eficazes para revelar a presença de defeitos, mesmo para programas pequenos (Barbosa et al., 2007).

Os critérios dessa categoria utilizam a análise do fluxo de dados do programa para derivar os requisitos de teste. Eles requerem que sejam testadas as interações que envolvem definições de uma variável e seus usos posteriores.

Uma família de critérios baseados em fluxo de dados é proposta por Rapps e Weyuker (Rapps e Weyuker, 1985). Para derivar os requisitos de teste dessa família foi proposta uma extensão do GFC, o **grafo Def-Uso**, na qual são adicionadas informações a respeito do fluxo de dados do programa.

Para obter o grafo Def-Uso a partir do GFC associa-se a cada nó i os conjuntos $c\text{-uso}(i)$ e $def(i)$, e a cada arco (i,j) o conjunto $p\text{-uso}(i,j)$, onde:

- $c\text{-uso}(i)$: conjunto de variáveis com c-uso global no bloco i ;
- $def(i)$: conjunto de variáveis com definições globais no bloco i ;
- $p\text{-uso}(i,j)$: variáveis com p-uso no arco (i, j) .

Além disso, foram definidos os conjuntos:

- $dcu(x,i)$: conjunto dos nós j , tal que $x \in c\text{-uso}(j)$, $x \in def(i)$ e existe um caminho livre de definição com relação a x do nó i para o nó j .
- $dpu(x,i)$: conjunto dos arcos (j, k) tal que $x \in p\text{-uso}(j,k)$, $x \in def(i)$ e existe um caminho livre de definição com relação a x do nó i para o arco (j, k) .

Ainda utilizando o código-fonte do programa listado na Seção 2.3, construiu-se um novo diagrama com as informações referentes ao fluxo de dados. O resultado é exibido na Figura 2.4.

Os critérios definidos por essa família são:

Figura 2.4: Grafo de fluxo de dados do método `calcValorLiquido()`

- **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. Dentre as variações desses critérios estão: critérios todos-p-usos, todos-p-usos/algum-c-uso e todos-c-usos/algum-p-uso.
- **Todos-du-caminhos:** requer que toda associação entre uma definição de variável e subsequentes p-usos ou c-usos dessa variável seja exercitada por todos os caminhos livres de definição e livres de laço que cubram essa associação.

A maior parte dos critérios baseados em fluxo de dados estabelece que para requerer um determinado elemento (caminho, associação, etc.) exista a ocorrência explícita de um uso de variável. Esse aspecto determina as principais limitações desses critérios e motivou a definição da família de critérios Potenciais Usos (Maldonado, 1991). Os critérios que fazem parte dessa família são:

Todos-potenciais-usos: requer para todo nó i e variável x para qual exista definição em i que pelo menos um caminho livre de definição com relação a x do nó i para todo nó j e para todo arco possível de ser alcançado a partir de i seja exercitado;

Todos-potenciais-usos/du: requer para todo nó i e variável x para qual exista definição em i que pelo menos um potencial-du-caminho ¹ com relação a x do nó i para todo nó e para todo arco possível de ser alcançado a partir de i seja exercitado;

Todos-potenciais-du-caminhos: requer para todo nó i e variável x para qual exista definição em i que todo potencial-du-caminho com relação a x do nó i para todo nó e para todo arco possível de ser alcançado a partir de i seja exercitado.

2.7 Geração de dados de teste

Conforme explicado na Seção 2.2, o principal objetivo do teste de software é detectar o maior número de defeitos possível. Para ganhar confiança suficiente de que a maioria dos defeitos é encontrada, idealmente o teste deveria ser exaustivo. Como isso não é possível na prática, foram estabelecidos técnicas e critérios de teste. Além disso, deve-se realizar uma escolha criteriosa dos casos de teste a fim de que eles tenham uma alta probabilidade de encontrar possíveis defeitos. Sendo assim, a geração de casos de teste corresponde a uma importante atividade dentro do Teste de Software. Um caso de teste consiste de dados de teste, ou seja, dados de entrada que serão executados no programa, e da saída esperada (Vergilio et al. (2007); McMinin (2004)).

Na indústria, a seleção de dados de teste geralmente é um processo manual. No entanto, essa prática é extremamente cara, difícil e trabalhosa. A automatização dessa tarefa tem sido limitada, pois não existe um algoritmo de propósito geral para determinar um conjunto de teste que satisfaça um critério (McMinin, 2004). Não é possível nem mesmo determinar automaticamente se esse conjunto existe. Dependendo do tamanho e da complexidade do software envolvido, a geração de dados de teste pode ser um problema indecível. Apesar das limitações existentes, encontram-se na literatura diferentes categorias de técnicas que podem ser utilizadas para gerar dados de teste para satisfazer os diversos critérios. As mais conhecidas são: geração aleatória de dados; geração com execução simbólica; geração com execução dinâmica; geração utilizando técnicas de computação evolutiva; e geração de dados sensíveis a defeitos (Vergilio et al., 2007).

McMinin (2004) divide ainda a automação da geração de dados de teste em quatro áreas: cobertura da estrutura específica do programa (remetendo aos critérios de teste estrutural), execução de características específicas do programa conforme descrito pela especificação (remetendo aos critérios de teste funcional), abordagem caixa-cinza e verificação de propriedades não-funcionais. As técnicas discutidas nesta seção se enquadram na primeira área e foram obtidas de Vergilio et al. (2007).

¹Análogo ao du-caminho, mas a ocorrência explícita do uso da variável não é requerida

2.7.1 Geração aleatória

Nessa técnica, os pontos do domínio de entrada são selecionados aleatoriamente até que o critério de teste seja satisfeito. A geração aleatória é prática, mais fácil de automatizar e menos custosa (McMinn, 2004). No entanto, gerar dados de teste aleatoriamente não garante a satisfação do critério e nem a seleção dos melhores pontos, ou seja, dos pontos com a maior probabilidade de revelar defeitos.

2.7.2 Geração com execução simbólica

Essa técnica não é a execução do programa propriamente dita, mas o processo de atribuir expressões às variáveis do programa à medida que um caminho segue a estrutura do código. O resultado é uma expressão simbólica chamada computação do caminho que expressa as variáveis de saída em termos das variáveis de entrada, e uma expressão chamada condição do caminho que expressa condições para o caminho ser executado.

Inicialmente, valores simbólicos são atribuídos às variáveis de entrada. Cada comando associado aos nós do caminho é avaliado em termos desses valores. Quando uma atribuição é executada, os valores do lado direito são substituídos pelos seus valores simbólicos. O resultado é simplificado e será o novo valor simbólico atribuído à variável que está do lado esquerdo da atribuição. Da mesma maneira, se um comando condicional é encontrado, o predicado associado é descrito por uma restrição em termos dos valores simbólicos das variáveis nele envolvidas. Essa restrição é chamada de interpretação do predicado.

A computação do caminho é obtida pela execução simbólica de todos os comandos que compõem o caminho, representada pelos valores simbólicos atribuídos às variáveis de saída. A condição do caminho é dada pelo conjunto de restrições associadas a todos os predicados encontrados ao longo do caminho. O domínio do caminho é dado pelo conjunto de valores que satisfazem a condição do caminho. O dado de teste é então gerado escolhendo-se um elemento desse conjunto. Nem sempre é possível determinar uma solução que satisfaça a condição do caminho, ou seja, determinar se o caminho é ou não executável.

Há algumas restrições ao se derivarem expressões simbólicas tais como laços, referências a vetores e ponteiros e chamadas de funções ou procedimentos.

2.7.3 Geração com execução dinâmica

Surgiu como uma alternativa à execução simbólica para solucionar problemas com arranjos e variáveis dinâmicas. Baseia-se na execução real do programa em teste, em métodos de minimização de funções e análise do fluxo de dados. Dados reais são atribuídos às variáveis de entrada e, ao executar o programa, os resultados são avaliados e melhorados

utilizando otimização de funções, que guiam a busca por dados adequados. Uma vez que as referências a variáveis compostas são conhecidas em tempo de execução, muitas das restrições da execução simbólica podem ser contornadas.

2.7.4 Geração com algoritmos evolutivos

Algoritmos evolutivos são métodos de busca e otimização que, aplicando as ideias da Teoria da Evolução de Darwin, simulam os processos naturais de evolução; os indivíduos mais adaptados de uma população têm maior probabilidade de sobreviver e, por isso, de passar aos seus descendentes seu material genético. A classe de algoritmos evolutivos mais conhecida é a dos algoritmos genéticos (AGs). Para utilizar algoritmos genéticos é necessário representar o espaço de busca do problema e suas possíveis soluções de forma computacional. Para implementar a seleção natural de forma que a população evolua para boas soluções é necessário uma medida que decida quão boa é uma dada solução. Essa medida pode ser uma função objetivo, que é um modelo matemático ou uma simulação de computador, ou pode ser uma função subjetiva em que humanos escolhem as melhores soluções sobre as piores. Em essência, a medida de *fitness* deve determinar a aptidão (*fitness*) relativa de uma dada solução candidata, que será utilizada pelo AG para guiar a evolução de boas soluções (Burke e Kendall, 2013; McMin, 2004).

OS AGs dependem de uma população de soluções candidatas. O tamanho da população, que geralmente é especificado por um parâmetro do usuário, é um dos fatores importantes que afetam a escalabilidade e a performance dos AGs. Por exemplo, uma população pequena pode conduzir a uma convergência prematura e produzir soluções inferiores. Por outro lado, uma população muito grande pode levar a gastos desnecessários de tempo computacional. Operadores inspirados pela seleção natural direcionam a busca pelos indivíduos mais promissores e operadores inspirados pela genética recombina e muta partes desses indivíduos a fim de descobrir os melhores. O Algoritmo 1 apresenta uma descrição em alto nível de um algoritmo genético, retirada de McMin (2004). Ele funciona da seguinte forma: Primeiramente, uma população de indivíduos é gerada aleatoriamente ou “semeada” com indivíduos pré-configurados (linha 1), no entanto, o conhecimento do domínio ou outras informações podem ser facilmente incorporadas na geração da população inicial. Os passos seguintes se repetem até que a solução desejada seja encontrada ou até que um número máximo de gerações seja alcançado (linha 2). A cada iteração desse ciclo, os indivíduos na população corrente são avaliados computando seus valores de *fitness* a fim de obter uma estimativa de suas capacidades em resolver o problema (linha 3). O próximo passo é realizar a seleção dos indivíduos que irão sobreviver para a próxima geração utilizando algum mecanismo de seleção (linha 4). A ideia da seleção é reproduzir os melhores e mais aptos indivíduos, no entanto, quanto maior for o viés em relação a eles

é possível que resulte em gerações futuras dominadas, reduzindo a diversidade e aumentando a chance de convergência prematura no espaço de busca. Por outro lado, quanto mais fraca for a estratégia é possível que resulte em muita exploração e uma evolução insuficiente para que a busca obtenha progressos substanciais. Uma vez que o conjunto de pais foi selecionado, a recombinação é aplicada com uma probabilidade P_c (linha 5). A recombinação considera dois indivíduos, atuando como pais, e combina suas partes formando duas novas soluções, ou seja *offsprings*, para tentar reproduzir melhores candidatos. Após a recombinação, um estágio de mutação é empregado, o qual é responsável por introduzir ou reintroduzir material genético na busca, para que a diversificação se mantenha (linha 6). Enquanto a recombinação opera em dois ou mais cromossomos dos pais, a mutação modifica uma solução localmente ou aleatoriamente. Finalmente, a população de pais original é substituída pela população *offspring* criada pela seleção, recombinação e mutação. Podem ser utilizadas algumas técnicas de substituição em algoritmos genéticos, como a substituição elitista (Burke e Kendall, 2013; McMinn, 2004).

Algorithm 1 Descrição em alto nível de um Algoritmo Genético (McMinn, 2004)

```

1: Gerar aleatoriamente ou semear a população inicial  $P$ 
2: repeat
3:   Avaliar a função de fitness de cada indivíduo em  $P$ 
4:   Selecionar os pais a partir de  $P$  de acordo com o mecanismo de seleção
5:   Recombinar os pais para formar o novo offspring
6:   Construir a nova população  $P'$  a partir dos pais e do offspring
7:   Mutar  $P'$ 
8:    $P \leftarrow P'$ 
9: until Atingir a condição de parada

```

Há várias técnicas para aplicação dos algoritmos genéticos que são classificados em duas linhas, Abordagens Orientadas a Cobertura e Abordagens Orientadas a Estrutura. As Abordagens Orientadas a Cobertura recompensam os indivíduos com base na estrutura do programa que foi coberta. Os indivíduos que executam caminhos mais longos são recompensados, e os que executam os caminhos já cobertos são punidos. No entanto, essa abordagem dificilmente atinge a cobertura total, uma vez que nada garante que trechos com execução restrita a valores específicos do domínio sejam executados (McMinn, 2004).

As Abordagens Orientadas a Estrutura exploram as informações nos predicados dos ramos e se subdividem em três categorias: 1) Orientadas a Distância do Ramo, em que se escolhe um caminho e os predicados dos ramos mais relevantes são extraídos do programa. O algoritmo genético é então utilizado para encontrar os dados de entrada que satisfazem todos os predicados dos ramos de uma vez. Nesse caso, a função objetivo corresponde à soma dos valores de distância dos ramos; 2) Orientadas a Controle, cuja função objetivo considera os nós da ramificação que precisam ser executados, a fim de exercitar a estrutura desejada. Para isso, podem-se utilizar um grafo de controle de dependências, que define a dependência entre todos os nós, informando que nós devem ser necessariamente executados

por desencadearem a execução dos demais; e 3) Combinação das duas abordagens, em que se identificam nós de controle de dependência da estrutura e, caso um indivíduo tome um ramo crítico a partir desses nós, um cálculo da distância é realizado com base no predicado do ramo requerido e do caminho alternativo que deve ser executado (McMinn, 2004).

2.7.5 Geração de dados sensíveis a defeitos

Procuram selecionar pontos do domínio que estão relacionados a certos tipos de defeitos e oferecem estratégias de geração de dados de teste que terão alta probabilidade de revelá-los. As principais estratégias de geração de dados de teste sensíveis a defeitos são: teste de domínios, teste baseado em restrições e teste baseado em predicados.

2.7.5.1 Teste de domínios

Desenvolvida com o propósito de selecionar dados de teste para um conjunto de caminhos de programas. No entanto, não especifica como os caminhos são selecionados, devendo então ser utilizado em conjunto com critérios baseados em caminhos (estruturais).

O objetivo é detectar defeitos de domínios, ou seja, defeitos que atribuem um domínio incorreto a um dado caminho, selecionando dados de teste no limite do domínio do caminho ou próximos dele. Essa técnica foi originalmente proposta por White e Cohen (White e Cohen, 1980) e analisa os limites do domínio de um caminho para gerar os dados de teste que o executam. O domínio do caminho é definido como o conjunto de dados de entrada que satisfazem a “condição do caminho”. Os limites do domínio do caminho são dados pelas condições associadas às arestas que compõem o caminho. A cada condição corresponde um predicado, que é uma combinação lógica de expressões relacionais.

A técnica de teste de domínios está baseada na análise geométrica dos limites do domínio. Tem como objetivo detectar erros nas bordas, dados por pequenas diferenças entre a borda correta e a borda do programa em teste (borda dada), embora o teste de domínios não exija que se tenha conhecimento do programa correto. Ela seleciona dois tipos de pontos de teste: pontos *on* que pertencem à borda dada e pontos *off* que ficam a uma distância muito pequena e devem pertencer ao domínio que não contém a borda. Se a borda é fechada, os pontos *on* pertencem ao domínio do caminho que está sendo testado e os pontos *off* pertencem a algum domínio adjacente. Se a borda é aberta, os pontos *on* pertencem a algum domínio adjacente, enquanto os pontos *off* pertencem ao domínio em teste.

White e Cohen (1980) propuseram a técnica *NX1* que requer que sejam escolhidos *N* pontos *on* e um ponto *off* para cada borda do domínio do caminho. Duas técnicas mais rigorosas são a *NXN*, que requer que sejam selecionados *N* pontos *on* e *N* pontos *off*,

e a VXV , em que V é o número de vértices da borda dada. Se existem V vértices, são escolhidos V pontos *on* e V pontos *off*, próximos aos vértices ou nos vértices.

As três técnicas são consideradas efetivas para detectar erros em bordas. A técnica NXN leva alguma vantagem sobre a $NX1$, por ser melhor na maioria dos casos. No entanto, a técnica VXV raramente seria melhor que a NXN , visto que os pontos por ela escolhidos formam um superconjunto dos escolhidos pela NXN . Entretanto, a técnica $NX1$ é mais prática e fácil de aplicar.

2.7.5.2 Teste baseado em restrições

Tem como objetivo auxiliar a geração de dados de teste adequados ao teste de mutação. Usa restrições algébricas projetadas para detectar um tipo particular de defeito no programa. O teste baseado em restrições utiliza os conceitos de análise de mutantes para gerar dados de teste. Conforme apresentado na Seção 2.5, para se matar um mutante são necessárias três condições: alcançabilidade, necessidade e suficiência.

A alcançabilidade refere-se a executar o comando alterado. Assim, devem-se gerar dados de teste que garantam alcançar o comando mutado. Para satisfazer a condição de necessidade, é necessário que, imediatamente após a execução de um comando mutado, o estado do programa mutante seja diferente do estado do programa original. Essa condição descreve uma restrição que deve ser satisfeita pelo dado de teste para que estados distintos nos programas mutante e original sejam produzidos. A condição de suficiência está relacionada ao fato de que a diferença produzida após a execução do comando mutado seja propagada até que comandos de saída sejam executados e resultados diferentes sejam produzidos, possibilitando assim a revelação do defeito. Derivar dados de teste que satisfaçam à condição de suficiência é impraticável e, em geral, não se consegue determiná-la completamente.

2.7.5.3 Teste baseado em predicados

O teste baseado em predicados requer que cada predicado (ou condição) em um programa seja testado. Os predicados são classificados em simples (variável booleana ou expressão relacional, com a possibilidade de usar um ou mais operadores NOT) e compostos (consiste de pelo menos um operador booleano binário (OR e AND), dois ou mais operandos e, possivelmente, operadores NOT e parênteses). Dessa forma, as técnicas baseadas em predicados são classificadas em dois grupos: as que testam predicados simples e as que testam predicados compostos.

Técnicas que consideram apenas predicados simples podem deixar de detectar erros de predicados em expressões booleanas. As técnicas que consideram predicados compostos propõem testes exaustivos para todas as combinações, o que pode ser inviável para

predicados com grande número de operadores. O problema com predicados compostos é que, se o teste exaustivo não for realizado, poderá acontecer que, embora os predicados individuais sejam avaliados incorretamente, o resultado da avaliação de toda a expressão seja coincidentemente correto. Dessa forma, Tai (1993) propôs as técnicas BOR (*Boolean Operator Testing*) e BRO (*Boolean and Relational Operator Testing*) para detectar defeitos em predicados compostos. A primeira garante a detecção de defeitos em operadores booleanos. Tai (1993) considera que se uma técnica é efetiva em determinar defeitos em operadores booleanos, ela também o será em determinar defeitos em expressões booleanas. A segunda técnica garante a detecção de defeitos em operadores booleanos e relacionais. As restrições geradas pelas técnicas BRO e BOR também podem ser utilizadas para gerar dados de teste adequados à análise de mutantes e as técnicas de teste estrutural e funcional.

2.8 Considerações finais

Neste capítulo foi apresentada uma revisão dos principais conceitos envolvendo teste de software que são relevantes para a proposta desta tese. Foram apresentadas as técnicas de teste funcional, baseada em defeitos e estrutural, com um enfoque maior na última. Também foram discutidos os critérios baseados em fluxo de controle e em fluxo de dados da técnica estrutural. Por fim, foram apresentados os principais conceitos sobre geração automática de dados de teste.

Teste estrutural de software embarcado com ênfase em veículos autônomos

3.1 Considerações iniciais

Este capítulo apresenta uma revisão da literatura sobre teste de software em sistemas embarcados com ênfase no teste estrutural e também com vistas a explorar como são conduzidos testes de veículos autônomos, que são um tipo de sistema embarcado. Vale ressaltar que optou-se por conduzir uma revisão de literatura em um tema mais amplo que veículos autônomos pois foram encontrados poucos trabalhos sobre esse assunto. Com o objetivo de obter um panorama mais abrangente e exploratório do estado da arte do teste estrutural nesse contexto, foi conduzido um mapeamento sistemático (Petersen et al., 2008) no primeiro semestre de 2010 e repetido no início de 2015. O processo de revisão bibliográfica e os principais resultados obtidos são relatados na Seção 3.2.

A partir desse primeiro conjunto de informações sobre o assunto, foram sendo conduzidas outras pesquisas bibliográficas mais informais, por meio de pesquisas nas principais bibliotecas digitais da área, por indicação de especialistas e por busca nas referências de artigos considerados importantes para esta tese. Alguns temas ganharam mais importância, como o uso de algoritmos evolutivos e baseados em busca para gerar dados de teste. Também procurou-se entender como ocorre o teste de veículos autônomos e foram analisados artigos e teses da área de robótica móvel/veículos autônomos. Na Seção 3.3 são

discutidas pesquisas relativas ao teste de software em sistemas embarcados com ênfase no teste estrutural. A maior parte dos artigos discutidos são provenientes do mapeamento sistemático, mas são incluídos também alguns que foram selecionados posteriormente. Em seguida, na Seção 3.4 são discutidas pesquisas que descrevem abordagens para geração de dados de teste. As pesquisas tratadas nestas duas últimas seções são mais gerais e não focam especificamente em veículos autônomos. Nelas também são apresentadas e discutidas ferramentas para teste estrutural e para geração automática de dados em sistemas embarcados. Por fim, na Seção 3.5 são apresentadas as considerações finais deste capítulo.

3.2 Mapeamento sistemático sobre teste estrutural em sistemas embarcados

De acordo com Petersen et al. (2008) e Kitchenham e Charters (2007) uma revisão da literatura na forma de um mapeamento sistemático é planejada para fornecer uma visão geral de uma área de pesquisa e para verificar se existem evidências de pesquisas sobre um assunto e produzir uma indicação da quantidade de evidências. Como parte desta tese foi conduzido um mapeamento sistemático usando um processo composto por três fases, a saber:

1. **Fase 1 - Planejamento:** nesta fase, os objetivos da pesquisa e o protocolo da revisão foram definidos. O protocolo é um plano pré-determinado que descreve as questões de pesquisa e como o mapeamento sistemático será conduzido.
2. **Fase 2 - Condução:** nesta fase, os estudos primários foram identificados, selecionados e avaliados de acordo com os critérios de inclusão e exclusão estabelecidos previamente. Para cada estudo selecionado, os dados foram extraídos e sintetizados.
3. **Fase 3 - Análise dos Resultados:** nessa fase, um relatório final foi escrito com base na análise dos estudos obtidos.

3.2.1 Fase 1 - Planejamento

Foram definidos os seguintes objetivos para o mapeamento sistemático:

- identificar e analisar os problemas relacionados ao teste estrutural em sistemas embarcados;
- identificar e analisar critérios de teste estrutural existentes para sistemas embarcados;

- identificar e analisar as ferramentas desenvolvidas e experimentos ou estudos de casos realizados para testes de sistemas embarcados;
- identificar e analisar as características do processo de desenvolvimento de sistemas embarcados e como elas afetam os testes.

Para atender aos objetivos propostos, foram elaboradas duas questões de pesquisa principais e três questões de pesquisa secundárias, cada uma com critérios próprios de inclusão e exclusão de trabalhos:

1. Questões Primárias

Q1 - Que problemas e dificuldades relacionadas ao teste estrutural de sistemas embarcados têm sido encontrados?

Q2 - Quais são os critérios existentes para teste estrutural de sistemas embarcados?

2. Questões Secundárias

Q3 - Quais são as características do processo de desenvolvimento de sistemas embarcados e como elas afetam os testes?

Q4 - Que ferramentas para teste estrutural de sistemas embarcados têm sido desenvolvidas?

Q5 - Que experimentos ou estudos de caso para testes estrutural de sistemas embarcados têm sido realizados?

Com o intuito de estabelecer a estratégia de busca e considerando as questões de pesquisa, foram inicialmente identificadas as principais palavras-chave que são “*structural testing*” e “*embedded systems*”. Em seguida, foram encontrados sinônimos em inglês para essas palavras-chaves:

- *Structural Testing*: (“*structural testing*” “*code-based testing*” “*white box testing*” “*coverage testing*” “*coverage-based testing*” “*code based testing*” “*white-box testing*” “*coverage based testing*”)
- *Embedded Systems*: (*embedded* “*unmanned vehicles*” “*unmanned autonomous vehicles*” *uav* “*unmanned ground vehicles*” *ugv* *robot* *robots*)

Vale ressaltar que a palavra-chave escolhida deve ser simples o suficiente para trazer muitos resultados e, ao mesmo tempo, rigorosa o suficiente para cobrir apenas o tópico da pesquisa desejado. Levando isso em consideração, na formação da string de busca foi considerada apenas a palavra *embedded* da palavra-chave *embedded systems*. Foi utilizado

CAPÍTULO 3. TESTE ESTRUTURAL DE SOFTWARE EMBARCADO COM ÊNFASE EM VEÍCULOS AUTÔNOMOS

o operador booleano OR para integrar os termos-chave e seus sinônimos e o operador AND para integrar os diferentes termos-chave. Dessa forma, a *string* final utilizada foi a seguinte:

("structural testing" OR "code-based testing" OR "white box testing" OR "coverage testing" OR "coverage-based testing" OR "code based testing" OR "white-box testing" OR "coverage based testing") AND (embedded OR "unmanned vehicles" OR "unmanned autonomous vehicles" OR uav OR "unmanned ground vehicles" OR ugv OR robot OR robots)

Além das questões de pesquisa e da estratégia de busca, também foram estabelecidas as fontes de dados (ou seja, base de dados eletrônicas) a serem utilizadas para encontrar os estudos primários. Os critérios utilizados para selecionar as fontes foram: atualização de conteúdo – publicações são regularmente atualizadas; disponibilidade – texto completo dos artigos estão disponíveis; e a qualidade dos resultados – a precisão dos resultados retornados pela pesquisa. As fontes selecionadas foram: *ACM Digital Library*, *IEEE Xplore*, *SpringerLink*, *Scopus* e *ISI Web of Knowledge*.

Outro elemento importante do planejamento do mapeamento sistemático é definir os Critérios de Inclusão e os Critérios de Exclusão. A partir desses critérios é possível incluir estudos primários que são relevantes para responder as questões de pesquisa e excluir os estudos que não as respondam. Assim, os seguintes critérios de inclusão de trabalhos foram definidos para atender a cada uma das questões de pesquisa:

- **C1** - O artigo descreve dificuldades relacionadas ao teste estrutural de sistemas embarcados.
- **C2** - O artigo descreve métodos e critérios existentes para teste estrutural de sistemas embarcados.
- **C3** - O artigo descreve características do processo de desenvolvimento de sistemas embarcados e como essas características afetam os testes.
- **C4** - O artigo descreve ferramentas para teste estrutural de sistemas embarcados.
- **C5** - O artigo descreve experimentos ou estudos de caso para testes estrutural de sistemas embarcados.
- **C6** - O artigo trata de geração automática de dados de teste baseada em teste estrutural.

Os critérios de exclusão estabelecidos foram:

- **E1** - Não trata de teste estrutural de software.

- **E2** - Não trata de sistemas embarcados.
- **E3** - Não se encontra redigido em inglês ou português.
- **E4** - A versão completa do artigo não está disponível.
- **E5** - Ano de publicação anterior a 1990.
- **E6** - Artigo duplicado.

3.2.2 Fase 2 - Condução

A fase de condução, ou execução, do mapeamento sistemático consiste da busca de estudos nas fontes definidas no planejamento. Ao passo que vão sendo obtidos os resultados dessas buscas, eles devem ser empacotados.

Tabela 3.1: Quantidade de trabalhos por categoria e fonte

Fonte/Categoria	ACM	IEEE	Scopus	Springer	ISI	Total/Cat.
Incluídos	16	14	27	3	15	75 (38,66%)
Excluídos	16	14	55	3	31	119 (61,34%)
Total/Fonte	32 (16,49%)	28 (14,43%)	82 (42,27%)	6 (3,1%)	46 (23,71%)	194 (100%)

Para dar suporte à organização dos estudos primários, foi utilizada uma planilha Excel. A documentação dos resultados das buscas foi inserida em uma planilha, com uma aba para cada fonte de busca. Como resultado, 194 publicações foram identificadas. O próximo passo foi realizar a leitura dos resumos dessas publicações e aplicar os critérios de inclusão e exclusão. Como resultado, foram incluídas 75 publicações (38,66%) e excluídas 119 publicações (61,34%). Entre as 194 publicações havia 83 repetidas (42,78%). Entre as 111 restantes, 30 foram incluídas e 81 foram excluídas. Na Tabela 3.1 apresenta-se o detalhamento da quantidade de publicações incluídas e excluídas por fonte de busca e na Tabela 3.2 apresenta-se o detalhamento da quantidade de publicações repetidas.

O processo da seleção preliminar foi documentado na mesma planilha citada acima. A Tabela 3.3 apresenta as 30 publicações incluídas e os critérios usados para inclusão. Pode-se observar que a maioria das publicações foi incluída pelo critério 1 (ou seja, estudos primários que descrevem dificuldades no teste estrutural de sistemas embarcados).

Tabela 3.2: Quantidade de publicações considerando as repetidas

Categoria	Repetidos	Não repetidos	Total
Publicações incluídas	45 (23,19%)	30 (15,46%)	75 (38,66%)
Publicações excluídas	38 (19,58%)	81 (41,75%)	119 (61,34%)
Total de publicações	83 (42,78 %)	111 (57,21%)	194 (100%)

Na Figura 3.1 é apresentado um gráfico que relaciona a quantidade dos artigos que foram incluídos por critério de inclusão.

CAPÍTULO 3. TESTE ESTRUTURAL DE SOFTWARE EMBARCADO COM ÊNFASE EM VEÍCULOS AUTÔNOMOS

Tabela 3.3: Publicações incluídas

Ref	Título	Autor	CI	Ano
1	An automatically-generated run-time instrumenter to reduce coverage testing overhead	J. Jenny Li, David M. Weiss, Howell Yee	C1	2008
2	Architectural design, behavior modeling and run-time verification of network embedded systems	Man-Tak Shing, Doron Drusinsky	C3	2007
3	SCCP/x: a compilation profile to support testing and verification of optimized code	Raimund Kirner	C1	2007
4	An Instrumentation-Based Approach to Controller Model Validation	Rance Cleaveland, Scott A. Smolka, Steven T. Sims	C1	2008
5	Model-Based Quality Assurance of Automotive Software	Jan Jürjens, Daniel Reiß David Trachtenherz	C3	2008
6	Evolutionary White-Box Software Test with the EvoTest Framework: A Progress Report	Hamilton Gross, Peter M. Kruse, Joachim Wegener, Tanja Vos	C1, C4, C5	2009
7	Security vs. Test Quality: Fully Embedded Test Approaches Are the Key to Having Both	Stephen Pateras	C2	2004
8	Coverage-Based Testing on Embedded Systems	X. Wu, J. Jenny Li, D. Weiss, Y. Lee	C1, C5	2007
9	Nonintrusive Black- and White-Box Testing of Embedded Systems Software against UML Models	P. Graf, K. D. Muller-Glaser, C. Reichmann	C3	2007
10	An industrial case study of structural testing applied to safety-critical embedded software	Jing Guan, Jeff Offutt, Paul Ammann	C1, C5	2006
11	Rapid embedded system testing using verification patterns	Wei-Tek Tsai; Lian Yu; Feng Zhu; Paul, R.	C1	2005
12	Enhanced software design reusability in safety-critical embedded applications using automated structural test generation	Morton, S.D.	C3	1999
13	A model for concurrent states and its coverage criteria	Kojima, H.; Kakuda, Y.; Takahashi, J.; Ohta, T.	C1, C2	2009
14	Applying Particle Swarm Optimization to Prioritizing Test Cases for Embedded Real Time Software Retesting	Hla, K.H.S.; YoungSik Choi; Jong Sou Park;	C1	2008
15	Supporting Tool for Embedded Software Testing	Koong, Chornng-Shiuh; Lai, Hung-Jui; Chang, Chih-Hung; Chu, William C.; Hsueh, Nien-Lin; Hsiung, Pao-Ann; Shih, Chihhsiong; Yang, Chao-Tung;	C4	2010
16	Temporal white-box testing using evolutionary algorithms	Al Moubayed, N., Windisch, A.	C2	2009
17	Creation and validation of embedded assertion statecharts	Drusinsky, D., Shing, M.-T., Demir, K.A.	C3	2006
18	Algorithm for calculating the branch routes of an assembly program in software coverage testing	Tang, K., Wang, W.-Y., Xiang, Y., Luo, G.-C.	C2	2005
19	Architectural Unit Testing	Scollo, G., Zecchini, S.	C1, C5	2005
20	A Survey of Coverage-Based Testing Tools	Qian Yang, J. Jenny Li, D. Weiss	C4	2006
21	A concept for testing robustness and safety of the context-aware behaviour of autonomous systems	Micskei, Z., Szatmári, Z., Oláh, J., Majzik, I.	C1, C3, C6	2012
22	Applying model-based testing to a train control system	Didrich, K., Herbst, S., Vieira, M.	C1, C3	2006
23	Critical path coverage testing of embedded software based on LCSAJ	Xiao Zhi ; Jianghua Zhang	C1	2014
24	Empirical evaluation on FBD model-based test coverage criteria using mutation analysis	Shin, D., Jee, E., Bae, D.-H	C1, C2, C5	2012
25	Research on instrumentation technology for the embedded assembly software testing based on host computer	Dong Ting ; Wang Zhongmin	C1	2011
26	Search based testing of embedded systems implemented in IEC 61131-3: An industrial case study	Doganay, K., Bohlin, M., Sellin, O.	C5, C6	2013
27	Selection and prioritization of test cases by combining white-box and black-box testing methods	Kukolj, S., Marinković, V., Popović, M., Bognár, S.	C6	2013
28	Software testing with code-based test generators: Data and lessons learned from a case study with an industrial software component	Braione, P., Denaro, G., Mattavelli, A., Vivanti, M., Muhammad, A.	C5, C6	2014
29	Test-case generation for embedded simulink via formal concept analysis	He, N., Rümmer, P., Kroening, D.	C6	2011
30	Automatic testing environment for multi-core embedded software - ATEMES	Koong, C.-S., Shih, C., Hsiung, P.-A., (...), Hsueh, N.-L., Yang, C.-T.	C4, C6	2012

3.2.3 Fase 3 - Análise dos resultados

Na Figura 3.2 é apresentado o número de estudos primários separados por ano, sendo que o mais antigo é de 1986. O trabalho mais antigo listado na Tabela 3.3 é de 1999,

Figura 3.1: Quantidade de publicações incluídas por critério de inclusão

uma vez que essa tabela contém apenas os trabalhos que foram incluídos pelos critérios de inclusão. Pela análise do gráfico, nota-se que houve um aumento do número de estudos primários relacionados ao teste estrutural de sistemas embarcados a partir de 2007 com um pico em 2011, indicando um aumento de interesse nesse tópico de pesquisa a partir dessa data. Como o mapeamento foi replicado nos primeiros meses de 2015, é possível que tenha trabalhos de 2014 que ainda não tenham sido indexados, não sendo então retornados na busca e, por isso, o gráfico mostra essa queda.

Figura 3.2: Estudos primários publicados por ano

Em relação à questão Q1 (Que problemas e dificuldades relacionadas ao teste estrutural de sistemas embarcados têm sido encontrados), foram identificados os seguintes problemas e dificuldades:

- A instrumentação pode levar à sobrecarga na execução do programa devido aos pontos de instrumentação inseridos. A sobrecarga pode alterar o comportamento da execução do programa ou impactar seu desempenho. Esse problema é ainda mais crítico para sistemas embarcados nos quais os recursos de memória e CPU são limitados (Li et al., 2008).

- Os programas no domínio de segurança crítica são otimizados durante a compilação, ou seja, o compilador pode introduzir novas decisões de fluxo de controle ao programa. Dessa forma, é necessário garantir que os casos de testes obtidos a partir do código fonte também sejam válidos no nível de código objeto (Kirner, 2007).
- Quase todos os sistemas embarcados contêm sensores e/ou atuadores e seus dados são cruciais para a operação de todo o sistema. Atrás desses atuadores e sensores há um ambiente que interage com o dispositivo embarcado. Sendo assim, o comportamento do sistema embarcado não depende somente dos eventos que ele recebe, mas também do estado percebido do ambiente (Graf et al., 2007; Micskei et al., 2012). Além disso, há um grande número de situações possíveis e podem ser requeridas adaptações para evoluir o contexto em que ele está inserido, fazendo com que o comportamento do sistema embarcado mude com o tempo baseado no *feedback* vindo do ambiente em evolução (Micskei et al., 2012).
- Em teste de sistemas de controle embarcado é muito difícil controlar todos os parâmetros que afetam o sistema. Além disso, limitações de hardware muitas vezes impedem que as variáveis atinjam determinados valores. Uma possível solução para esses dois problemas é utilizar simulação (Gross et al., 2009).
- A maioria dos sistemas embarcados utiliza software multitarefas e de tempo real em que as tarefas estão competindo por recursos devido a execução concorrente ocasionando condições de corrida e entrelaçamentos das sentenças das tarefas. Além disso, questões de entrelaçamento, *deadlines*, etc. precisam do reteste das unidades de software para seguir com as mudanças subsequentes (Didrich et al., 2006; Graf et al., 2007; Hla et al., 2008).
- As ferramentas de testes tradicionais têm foco nas estações de trabalho. Em termos de desempenho de teste, como o software embarcado executa na plataforma alvo, é mais difícil ser monitorado pelo engenheiro de teste, fazendo com que seja mais difícil aumentar o desempenho durante o processo de teste. Além disso, há a tarefa de testar o sistema em ambos ambientes reais e simulados, o ambiente real nem sempre está disponível ou os testes nesse ambiente podem ser muito caros (Didrich et al., 2006; Koong et al., 2010, 2012; Ting e Zhongmin, 2011).
- A utilização de modelos (Simulink, por exemplo) ou de técnicas como linhas de produtos no desenvolvimento do sistema embarcado pode dificultar a atividade de teste (He et al., 2011).

Em relação à pergunta Q2 (Quais são os critérios existentes para teste estrutural de sistemas embarcados), um trabalho define critérios de teste estrutural baseados em lógica, chamados de CACC, mas eles não diferem dos critérios para softwares tradicionais.

Embora não esteja diretamente relacionado ao teste estrutural, Shin et al. (2012) definiram três critérios de teste de cobertura para modelos baseados em FBD (*Function Block Diagram*): Cobertura Básica, Cobertura de Condição de Entrada e Cobertura de Condição Complexa.

Em relação à pergunta Q3 (Quais são as características do processo de desenvolvimento de software embarcado e como essas características afetam os testes), encontrou-se o seguinte:

- Desenvolvimento Dirigido por Modelos. Em um dos trabalhos, os autores propuseram uma abordagem para que também pudessem fazer o teste do código, a configuração e a instrumentação necessária do sistema a partir de diagramas de sequência da UML (Graf et al., 2007).
- Os desenvolvedores geralmente especificam sistemas embarcados utilizando cenários e um sistema de porte médio típico possui centenas de milhares de cenários. Cada cenário requer um script de teste e cada script de teste é potencialmente um programa trivial que requer o desenvolvimento individual e de depuração. Para resolver esse problema, um dos trabalhos propõe uma abordagem de padrão de verificação que classifica os cenários do sistema em padrões. Como a maioria das empresas de sistemas embarcados utiliza linha de produtos para desenvolvimento, com essa abordagem os testes seriam mais eficazes.

Em relação à pergunta Q4 (Que ferramentas para teste estrutural de sistemas embarcados têm sido desenvolvidas), foram encontradas as seguintes ferramentas:

- O framework “Teste RealTime” fornecido pela IBM Rational que faz o cálculo da cobertura de código obtida pelos casos de testes (Jürjens et al., 2008).
- O framework EvoTest que é composto por uma ferramenta de teste estrutural capaz de gerar casos de testes para funções em ANSI C, por um componente de teste caixa-preta e por um componente gerador de sinais (Gross et al., 2009).
- A ferramenta eXVantage, que é na verdade um conjunto de ferramentas de teste de cobertura, cujo objetivo é ajudar o desenvolvedor a aumentar a produtividade do teste. O conjunto de ferramentas é composto por quatro componentes principais: 1) inspetor automático de código; 2) módulo de instrumentação de programa que também inclui uma função para recuperação da estrutura do programa; 3) plataforma para execução de testes automáticos e 4) módulo de análises (Wu et al., 2007).

- A ferramenta ATEMES que apoia o cross-testing. Essa ferramenta reduz a sobrecarga de execução a partir das funções de teste implementadas nas plataformas embarcadas e diminui os esforços para o engenheiro de teste (Koong et al., 2010, 2012).

Em relação à pergunta Q5 (Que experimentos ou estudos de caso para teste estrutural de sistemas embarcados têm sido realizados), foram identificadas várias publicações em que os autores usaram experimentos ou estudos de caso para validar as propostas específicas relatadas na publicação. Algumas publicações relataram experimentos mais gerais:

- Proposta da abordagem SNB (*Super Nested Block*) para diminuir o *overhead* causado na instrumentação. Os autores realizaram um estudo de caso para medir o *overhead* causados pelos pontos de prova introduzidos na instrumentação seguindo a abordagem SNB proposta no artigo (Li et al., 2008; Wu et al., 2007).
- Comparação de abordagens tradicionais de teste com a verificação de modelos em relação ao número de defeitos encontrados (Jürjens et al., 2008).
- Condução de quatro estudos de caso, em que cada um consistia de um módulo de software embarcado da área automotiva com o objetivo de avaliar a ferramenta de teste estrutural proposta com o intuito de saber se é possível gerar casos de testes para as funções contidas nos estudos de casos (ou seja, se a ferramenta conseguiria lidar com a sintaxe e a semântica do código fonte) e também se a ferramenta iria completar os testes e exibir um resultado (Gross et al., 2009).
- Um dos artigos apresenta um estudo de caso para teste de software com características de tempo real embarcado em uma aplicação de segurança crítica da indústria de transportes com o objetivo de comparar dois conjuntos de testes em que um deles era constituído de testes manuais criados por engenheiros experientes da companhia e o outro conjunto era constituído de testes gerados a partir do código para satisfazer o critério baseado em lógica (CACC - *Correlated Active Clause Coverage*) (Guan et al., 2006).

3.2.4 Discussão

Os estudos primários encontrados foram na sua maior parte relacionados ao primeiro critério de inclusão, ou seja, das dificuldades relacionadas ao teste estrutural de sistemas embarcados. Por meio da leitura completa dos trabalhos, nota-se que a resposta à segunda questão primária é que não foi encontrada nenhuma proposição de critério de teste

específico para o teste de software embarcado. Os critérios e técnicas utilizados até então são os mesmos dos definidos para os softwares que executam em computadores de propósito geral.

Outro ponto observado é que as pesquisas relacionadas ao teste estrutural de sistemas embarcados, em sua maioria, tratam de estudos de caso para uma aplicação específica em que se tenta verificar como realizar o teste considerando as restrições e dificuldades do sistema em estudo. Entre essas restrições, podem-se citar limitações de memória, CPU, desempenho, concorrência, limitações de tempo, etc.

Uma parte considerável das ferramentas desenvolvidas está preocupada com a sobrecarga de execução causada pelo teste. Boa parte das ferramentas desenvolvidas não difere das ferramentas para teste de outros tipos de software.

Há poucos trabalhos relacionados às características de desenvolvimento de software embarcado que tratam especificamente de testes. Entre os trabalhos encontrados, nota-se que a maior parte deles se refere a desenvolvimento dirigido por modelos e de linha de produto de software.

3.3 Teste estrutural em sistemas embarcados

A maior parte dos estudos encontrados aplicam os critérios de teste estrutural de softwares tradicionais em softwares embarcados, realizando estudos de casos para avaliar a eficácia da aplicação dessas abordagens nesse tipo de sistemas (Gross et al., 2009; Guan et al., 2006; Jürjens et al., 2008). Também foram conduzidos estudos que aplicam as ferramentas desenvolvidas para o domínio de sistemas embarcados (Frescale, 2015; Gross et al., 2009; Koong et al., 2010, 2012; Wegener et al., 2002; Wu et al., 2007). Esses assuntos são discutidos com mais detalhes nas subseções seguintes.

Como há tipos de sistemas embarcados que possuem poucos recursos de memória, alguns trabalhos preocupam-se com o *overhead* causado pela instrumentação do programa. De acordo com o *survey* realizado por Yang et al. (2006), as ferramentas de teste de cobertura possuem um *overhead* de mais de 30%. Sendo assim, alguns estudos criaram abordagens para reduzi-lo (Li et al., 2008; Wu et al., 2007). Li et al. (2008) apresentam o método *Super Nested Block* (SNB) que seleciona pontos para instrumentação levando em conta o relacionamento de dependência de cobertura entre os blocos aninhados em arestas de super blocos. Um super bloco inclui instruções que são garantidamente executadas se uma delas for executada. Blocos aninhados são blocos dentro de arestas que estão aninhadas em pontos de ramificação de um super bloco. O método SNB é baseado no teorema que tem a seguinte hipótese: “A cobertura de um super bloco é determinada pela cobertura de seus blocos aninhados” (Li et al., 2008). Com o método SNB é possível

controlar a análise de fluxo para selecionar menos pontos de instrumentação, gerar o código instrumentado automaticamente e remover os pontos de verificação da instrumentação à medida que forem cobertos. Foram realizados experimentos em um sistema embarcado que mostrou que a abordagem proposta tem um desempenho suficientemente bom (Wu et al., 2007).

Em relação aos veículos autônomos, poucos estudos levam em conta o código fonte do veículo para definir os testes (Bacha et al., 2008a; Berger, 2013; Nothdurft et al., 2011; Rauskolb et al., 2008). Berger (2013) utiliza a análise de fluxo de controle para identificar dependências entre cenários e mudanças ocorridas no código fonte para determinar quais cenários deveriam ser re-executados por meio de simulação. Trepagnier et al. (2006), Nothdurft et al. (2011) e Rauskolb et al. (2008) criaram testes unitários durante o processo de desenvolvimento do software do veículo e os utilizaram para detectar possíveis defeitos ao realizar mudanças no sistema. Esses casos de testes são criados manualmente e sem levar em conta nenhum critério de teste estrutural. A fim de cobrir todas as sentenças, em Rauskolb et al. (2008) a cobertura do código fonte foi obtida contando automaticamente todas as sentenças executadas e gerando manualmente casos de testes para aumentar a cobertura.

3.3.1 Estudos de caso

Guan et al. (2006) realizaram um estudo de caso considerando um sistema de controle crítico da indústria de transportes e cujo sistema sob teste era formado por uma coleção complexa de máquinas de estado e algoritmos que interagiam entre si. Nesse estudo de caso foram comparados os testes que eram realizados na própria companhia (testes funcionais realizados de forma manual - MFT) com os critérios de teste CACC (*Correlated Active Clause Coverage*), que tem como objetivo testar cláusulas individuais dentro de expressões lógicas. Nesse critério, as expressões lógicas são formalizadas matematicamente utilizando os conceitos de predicado (que representa uma expressão que avalia um valor booleano) e cláusula (que representa um predicado que não contém nenhum operador lógico) a fim de introduzir critérios de cobertura com expressões lógicas mais simples. As expressões lógicas podem ser derivadas a partir do código fonte, da especificação ou de máquinas de estados finitos. Embora os predicados obtidos no estudo de caso não continham muitas cláusulas, os autores chegaram à conclusão de que o estudo trouxe evidências de que os critérios de teste estruturais podem ser úteis e compensadores para softwares embarcados e de segurança crítica.

Gross et al. (2009) avaliam a aplicabilidade do protótipo de uma ferramenta de teste estrutural evolutivo na indústria. Para isso, foram selecionados quatro estudos de caso onde cada um deles consiste de um módulo de software embarcado do setor automotivo

implementados em ANSI C, cujos códigos fontes vieram de duas ferramentas diferentes de geração automática de código. Essa ferramenta faz parte do framework EvoTest (ETF), que também é composto por um componente de teste caixa-preta e um componente para geração de sinal. No estudo de caso, uma empresa chamada BMS (Berner & Mattner Systemtechnik GmbH) avaliou a ferramenta em quatro módulos de software existentes com o objetivo de verificar se era possível gerar casos de teste para as funções contidas no estudo de caso e também se a ferramenta iria completar seu teste e apresentar um resultado. Os resultados mostraram que os casos de testes puderam ser gerados com sucesso para 37% das funções selecionadas para avaliação, o que, segundo os autores, foi um bom resultado para um protótipo; no entanto, ainda é necessário um trabalho significativo antes que essa abordagem seja aplicada na indústria.

Jürjens et al. (2008) comparam os testes tradicionais de software (incluindo teste estrutural) com a abordagem de verificação por modelos. Para realizar essa comparação, foi feito um estudo de caso no contexto da indústria automobilística a partir do estudo do desenvolvimento de uma unidade controladora de porta baseado em uma especificação textual disponível publicamente. Os componentes dessa unidade controladora foram modelados por duas ferramentas, a ASCET e a AutoFocus. A ASCET oferece suporte ao processo de desenvolvimento de software integrado de acordo com o modelo V. Em conjunto com o ASCET foram utilizadas ferramentas adicionais, como a CTEXL, usada para especificar graficamente os casos de teste na forma de uma matriz, e o framework “*Test Real Time*”, da IBM Rational, usado para medir a cobertura do código obtida pelos casos de teste. A ferramenta AutoFocus é baseada no conceito da metodologia Focus de descrição formal. Foram avaliados o número e os tipos de defeitos que foram encontrados e comparados ao esforço e eficácia dos métodos de garantia de qualidade. Como resultado, obteve-se que para a distribuição do esforço, os testes convencionais requereram a maior parte do esforço na avaliação dos resultados, no entanto, esse tempo poderia ser reduzido se houvesse ferramentas apropriadas disponíveis. Para a abordagem de verificação por modelos, a maior parte do esforço foi empregado na formulação das propriedades a serem verificadas. Segundo os autores, de modo geral, a verificação por modelos necessita de mais esforço de tempo comparada aos testes convencionais, mas esse esforço vale a pena pois alguns tipos de defeitos só poderiam ser encontrados por essa abordagem, especialmente os defeitos em que os desenvolvedores não podem considerar durante o teste estrutural, como por exemplo, defeitos de sincronização ou condições de limite. Para eles, a combinação dos dois métodos é o ideal, visto que ambos têm seus pontos fortes na descoberta de diferentes tipos de defeitos.

3.3.2 Ferramentas

Foram encontradas na literatura poucas ferramentas específicas ao teste estrutural em sistemas embarcados. Yang et al. (2006) apresentam um *survey* de ferramentas para o teste baseado em cobertura. Esse *survey*, embora não trate exclusivamente de sistemas embarcados, apresenta algumas ferramentas com foco em reduzir o *overhead* de tempo de execução, como por exemplo a ferramenta CodeTEST (Frescale, 2015).

Em relação à instrumentação de código-fonte, há as ferramentas ATEMES (*Automatic Testing Environment for Multi-core Embedded Software*), o framework exVantage e o conjunto de ferramentas “*Test Set Editor*” (TSE). A primeira, além de instrumentar automaticamente o código sob teste, gera os *drivers* e casos de teste e oferece suporte ao teste de unidade e cobertura. O teste pode ser realizado tanto nos ambientes *host-side* (onde são realizadas as ações de pré-processamento) e *target-side* (onde o software embarcado é implantado). O mecanismo de cobertura é realizado baseando-se nas funções de teste de cobertura do GCC. Foram realizados experimentos com alguns programas de estruturas de dados e os resultados mostraram que o ambiente de teste construído é viável. Todavia, para gerar os dados de teste, os autores utilizaram a abordagem aleatória e, além disso, não foram realizados experimentos com softwares embarcados reais (Koong et al., 2010, 2012).

O framework exVantage é um conjunto de ferramentas de teste de cobertura composto por quatro módulos principais. Além do módulo de instrumentação do programa (que também inclui uma função para recuperação da estrutura do programa), ele é composto por: inspetor automático de código, plataforma para execução de testes automáticos e módulo de análises (Wu et al., 2007). O TSE foi projetado para softwares escritos na linguagem ADA. Inicialmente, os engenheiros de teste constroem manualmente os casos de teste para o módulo a ser testado em uma planilha. A partir dessa planilha, são gerados automaticamente os *drivers* e *stubs* e, em seguida, o programa é compilado e os casos de teste executados. Dessa maneira, o programa é instrumentado e a cobertura alcançada é calculada. Segundo os autores, esse processo pode ser utilizado para se obter a certificação RTCA DO-178B.

Em relação a ferramentas de geração de casos de teste, além da ferramenta ATEMES, descrita anteriormente, há o framework EvoTest utilizado por Gross et al. (2009) e o ambiente para geração de dados de testes evolutivos desenvolvido por Wegener et al. (2002). O framework EvoTest é composto por uma ferramenta de teste estrutural capaz de gerar casos de testes para funções em ANSI C, por um componente de teste caixa-preta e por um componente gerador de sinais. O segundo contém seis componentes: 1) *parser* para análise dos objetivos de teste; 2) interface gráfica do usuário para especificação do domínio de entrada e dos objetivos de teste; 3) instrumentador; 4) gerador de *drivers*

CAPÍTULO 3. TESTE ESTRUTURAL DE SOFTWARE EMBARCADO COM ÊNFASE EM VEÍCULOS AUTÔNOMOS

de teste; 4) controlador de testes, que é responsável pelo gerenciamento dos objetivos de teste parciais com status em processamento, pela coleta de valores iniciais adequados para otimização dos objetivos parciais e pela gravação desses objetivos de testes parciais; e 5) conjunto de ferramentas de algoritmos evolutivos para geração de dados de teste. Foram conduzidos experimentos com exemplos reais utilizando essa ferramenta obtendo bons resultados e alcançando uma alta ou completa cobertura.

Para realizar o cálculo da cobertura de código obtida, além da ferramenta ATEMES, há também o framework “*Test RealTime*”, fornecido pela IBM Rational.

A ferramenta PATeCa implementada nesta tese oferece suporte ao gerenciamento dos testes de campo conduzidos com o veículo autônomo. Dessa forma, todas as informações relacionadas aos testadores, programas e versões, cenários de teste e *logs* gerados pela execução do teste de campo em um determinado cenário são gerenciados pela ferramenta. Além disso, ela permite realizar a reprodução dos *logs* obtidos no teste de campo e calcular a cobertura obtida considerando os critérios de teste estrutural todos-nós e todas-arestas. A PATeCa também é capaz de determinar quais trechos do *log* obtiveram um acréscimo da cobertura, chamados de trechos discriminatórios, possibilitando agrupar esses trechos considerando uma determinada heurística. Esses trechos discriminatórios podem ser utilizados em algoritmos de geração automática de dados de teste. Mais informações sobre a ferramenta PATeCa podem ser obtidas nos Capítulos 5 a 8 desta tese.

A Tabela 3.4 apresenta um resumo das ferramentas e suas funcionalidades.

Tabela 3.4: Ferramentas para teste estrutural em sistemas embarcados

Funcionalidades/ Ferramentas	ATEMES	exVantage	EvoTest	Wegener et. al (2002)	Test Real Time	PATeCa
Instrumentação	X	X	–	–	–	X
Geração de dados de teste	X	–	X	X	–	X
Inspeção de Código	–	X	–	–	–	–
Execução de Testes Automáticos	–	X	–	X	–	X
Cálculo de Cobertura	X	–	–	–	X –	X
Diminuição do <i>overhead</i>	–	X	–	–	–	–
Gerenciamento de testes	–	–	–	–	–	X
Redução dos dados de entrada	–	–	–	–	–	X
Reprodução <i>offline</i>	–	–	–	–	–	X

3.3.3 Outros trabalhos

Graf et al. (2007) descrevem uma abordagem para executar e especificar testes em modelos UML de casos de teste e que oferece suporte tanto ao teste funcional quanto ao teste estrutural. Nessa abordagem, os casos de teste são modelados utilizando casos de uso e diagramas de sequência. A partir desses modelos, os casos de teste podem ser gerados e,

mais tarde, avaliados com a resposta real do sistema do software embarcado. Nessa mesma linha, Drusinsky et al. (2007) apresentam um processo iterativo para o desenvolvimento de asserções de modelos de estado e utilizam essas asserções para verificar o projeto do protótipo desses modelos no início do processo de desenvolvimento de software. O processo proposto consiste das seguintes fases: 1) escrever especificações formais utilizando asserções de modelos de estado; 2) simulação baseada no JUnit e validação das asserções dos modelos de estado; 3) simulação baseada no JUnit e teste dos protótipos dos modelos ampliados com suas asserções; 4) teste estrutural automático baseada no JUnit dos modelos e ampliados com as asserções desses modelos; e 5) ajuste espiral do modelo e especificação usando os resultados do teste.

Em sistemas embarcados, obter a certificação do software é de suma importância. Nesse sentido, Morton (1999) apresenta quais critérios do teste estrutural são necessários pela certificação RTCA/DO - 178B e como esses critérios podem ser aplicados e implementados. Além disso, os autores discutem os aspectos da certificação do teste de software que podem ser automatizados, com ênfase no efeito causado pela automatização da reutilização do projeto. Esses aspectos, além de serem pertinentes a todos os programas compatíveis à certificação RTCA/DO - 178B, também podem ser utilizados às aplicações não certificáveis.

Outro trabalho relacionado ao teste estrutural é o de Kirner (2007). Segundo ele, alguns programas de domínio de segurança crítica são otimizados durante a compilação e, com isso, o compilador pode introduzir novas decisões no fluxo de controle do programa. Dessa forma, o autor apresenta em seu trabalho a questão de como garantir que a preservação da cobertura estrutural, conseguida com os casos de testes obtidos a partir do código fonte, seja válida também no código objeto. Para isso, foram analisadas as propriedades dos programas (como cobertura de sentença, de condição e de decisão) que precisariam ser mantidas para preservar diferentes coberturas de teste estrutural. Baseado nisso, é descrito um perfil de compilação que pode ser integrado em um compilador, o que garante que os critérios de cobertura estrutural são preservados durante a otimização da compilação.

Zhi e Zhang (2014) propõem uma abordagem que seleciona caminhos críticos a partir de vários caminhos do programa. Com isso, é possível balancear os custos do teste de software e simplificar o processo de projeto de casos de teste para cobertura de caminhos.

Shin et al. (2012) introduz os critérios de cobertura para FBD baseados nos critérios de teste estrutural. FDB é uma linguagem gráfica, destinada a controladores lógicos programáveis (PCL) e que tem sido muito utilizada para implementar software de segurança crítica tais como software de proteção de reator nuclear. Foram definidos três critérios de teste de cobertura para modelos baseados em FBD: Cobertura Básica, Cobertura de

Condição de Entrada e Cobertura de Condição Complexa. Os critérios de cobertura propostos são úteis nos aspectos de refletirem as características de fluxo de dados centrais de FBD e dar aos testadores conceitos intuitivos de cobertura estrutural. No entanto, foram conduzidos experimentos que descobriram que os critérios de cobertura FBD são bastante fracos para descobrir tipos específicos de falhas tais como lógica substituída ou blocos relacionais. No entanto, essas descobertas fornecem informações úteis para melhorar os critérios de cobertura existentes e as estratégias de teste e os autores já estão conduzindo pesquisas para desenvolver critérios de cobertura mais sofisticados e mais fortes para modelos FBD.

3.4 Geração de dados de teste em sistemas embarcados

Há um conjunto muito grande de artigos que discutem o uso de algoritmos baseados em busca, como os algoritmos genéticos ou evolutivos, no contexto de geração de casos de teste. Nessa área há excelentes revisões bibliográficas já publicadas, como as McMinn (2004) e de Tracey et al. (2002). Nesta seção são discutidos alguns artigos nesse contexto mas que têm foco em sistemas embarcados e que foram selecionados durante o mapeamento sistemático ou posteriormente.

O trabalho de Wegener et al. (1997) investiga a eficácia de algoritmos genéticos para validar a correção temporal de sistemas embarcados. A conformidade com as restrições de tempo para sistemas de tempo real é verificada estabelecendo tempos de execução máximo e mínimo. A função de *fitness* definida para o algoritmo genético utiliza o número de ciclos do processador. Os casos de teste foram obtidos a partir de uma ferramenta de árvore de classificação e utilizados como população inicial para dois softwares gráficos. A geração aleatória foi utilizada para o restante. Os algoritmos genéticos foram aplicados duas vezes, uma para encontrar o maior tempo de execução e outra para encontrar o menor. Os algoritmos genéticos conseguiram prever o maior e o menor tempo de execução de uma série de programas e superaram a geração aleatória, descobrindo soluções mais extremas mesmo que em determinadas ocasiões tenham requerido mais testes.

O trabalho de Nilsson e Henriksson (2005) apresenta uma extensão da ferramenta TrueTime, utilizada para cossimulação de tempo real, a fim de oferecer suporte à geração de casos de teste. A automação da geração de casos de teste é baseada na técnica de teste baseada em mutação. A principal ideia por trás dessa técnica é “adivinhar” sistematicamente quais falhas um projeto de tempo real contém e, então, avaliar quais seriam os piores efeitos de tais falhas. Na geração de casos de teste baseados em mutação, as entradas são a especificação do tempo real e os critérios de teste; esses critérios especificam os operadores de mutação que serão usados para criar os mutantes e, assim, determinar

os casos de teste que serão produzidos; um gerador de mutantes é aplicado aos operadores de mutação e envia as especificações mutantes para um analisador de execução, determinando se e como uma mutação pode levar a uma falha *timeless* (capacidade do software para atender às restrições de tempo). Uma vantagem na utilização dos critérios de teste baseados em mutação é que o esforço de teste pode ser estimado e quantificado pelo número de mutantes malignos (mutantes que contém falhas com más consequências).

O trabalho de Tlili et al. (2006) propõe uma abordagem para testar as restrições temporais em sistemas de tempo real (ERTT). Nessa abordagem, o objetivo do teste é transformado em um problema de otimização e um algoritmo evolutivo é utilizado para tentar resolver esse problema. O espaço de busca é formado pelo domínio de entrada do objeto de teste (dados de parâmetros e variáveis globais) e a função objetivo é baseada no tempo de execução necessário para executar o objeto de teste com os respectivos dados de entrada. A função ou é maximizada para a busca do pior tempo de execução (WCET) ou é minimizada para a busca do melhor tempo de execução (BCET). A população inicial do algoritmo evolutivo é gerada aleatoriamente e a busca termina se não há melhora nos tempos de execução extremos ou se um certo número (parametrizável) de gerações foi atingido. Os autores propõem dois métodos para encontrar os tempos de execução mais longos e mais curtos com o menor número de gerações. O primeiro é baseado na sementeira do algoritmo evolutivo com dados de teste estruturais que ajudam a busca evolutiva a explorar novas regiões no espaço de busca. O segundo método é baseado na ideia de que a maioria das variáveis globais e de entrada não são utilizadas em todos os seus tipos de dados. Ao restringir a gama dessas variáveis na população inicial, o processo de otimização é guiado para as regiões no espaço de busca, onde a gama de variáveis é realmente utilizada. Ao fazer isso, mais código é executado e, portanto, a segurança nos resultados entregues pelo ERTT é reforçada. Os autores também apresentam um método que restringe o alcance das variáveis de entrada na população inicial, melhorando o método a fim de reduzir o espaço de busca. Foram conduzidos experimentos que demonstraram um aumento na confiabilidade em termos de tempo de execução extrema e um ganho na eficiência em termos do número de gerações necessárias.

O trabalho de Windisch (2010) apresenta um método para automatizar o processo de geração de dados de teste que visa cobrir diagramas de estado StateFlow estruturalmente, garantindo a geração de dados adequados. Para isso, os princípios do teste estrutural evolutivo foram adaptados tanto para aplicação de diagramas StateFlow quanto para a consideração de sinais contínuos. O *workflow* para geração e automação dos sinais funciona da seguinte forma: o usuário fornece informações sobre a interface do sistema sob teste em termos da especificação de sinais descrito (1). Com base nisso, o gerador de sinais cria a especificação da sequência de otimização e informa a máquina de busca (2),

que fornece um conjunto inicial de soluções de problemas possíveis no retorno (3). Esse conjunto é utilizado para gerar as sequências de simulação (os sinais de entrada reais) para execução do sistema sob teste (4). Observações feitas durante a execução (5) são utilizadas para calcular o valor de *fitness* que alimenta de volta a máquina de busca (6). A máquina de busca utiliza essas avaliações a fim de gerar um novo conjunto de soluções de proposição. Esse *workflow* é realizado automaticamente para cada objetivo de teste decorrente do critério de cobertura selecionado.

Foi desenvolvido um ambiente estrutural de teste que implementa essa abordagem. Esse ambiente foi utilizado para conduzir experimentos com os objetivos de teste de um complexo diagrama de estado StateFlow industrial. O objetivo era gerar dados de teste realísticos e dependentes do contexto, alcançando a cobertura do estado para os diagramas de estado sob teste. Os objetivos de teste foram bem sucedidos para todos os estados viáveis.

Doganay et al. (2013) apresentam um estudo de caso para geração de teste baseado em busca para sistemas embarcados de unidades desenvolvidas utilizando FBD. Foram considerados 279 componentes diferentes a partir dos sistemas de controle de um trem desenvolvido pela Bombardier Transportation, um grande fabricante de veículo ferroviário. O software é compilado em código C com estrutura particular. É utilizado um algoritmo modificado de *hill climbing* para geração dos dados de teste para maximizar a cobertura MC/DC das atribuições com expressões lógicas no código C, mantendo a semântica da implementação original FBD. Foi realizada uma avaliação experimental para comparar a taxa de cobertura e o número necessário de execuções do algoritmo *hill climbing* com o teste aleatório. Os resultados mostraram que os testes aleatórios funcionam bem para a maioria das unidades sob teste, enquanto cerca de 30% se beneficiaram significativamente com o algoritmo de subida.

3.4.1 Redução da população inicial

Uma linha de pesquisa visando melhorar a eficácia dos algoritmos baseados em busca é a redução do domínio de entrada, isto é, das variáveis usadas como população inicial na estratégia de busca. Harman et al. (2007) estudaram o efeito da redução do domínio de entrada e apresentaram resultados da aplicação de algoritmos de busca a programas reais. Para os algoritmos evolutivos eles encontraram evidências empíricas de que a redução do domínio melhora a eficiência da busca, mas os resultados quanto à eficácia não foram conclusivos.

A linha de pesquisa sobre redução de suítes de casos de teste tem semelhanças com o estudo de população inicial realizado nesta tese. Yu et al. (2008), por exemplo, aplicam algoritmos clássicos (Harrold et al., 1993) de redução de casos de teste que se baseiam em

requisitos de teste – comandos executados, por exemplo – e depois usam essa informação para apoiar a localização de defeitos no código. Uma diferença importante em relação a esta tese é que geralmente não existe um conjunto de casos de teste preparado a priori, antes do teste de campo, e o objetivo da redução apresentada é outro.

Gordon e Arcuri (2012) discutem estratégias de “semeadura no teste” de software baseado em busca. Uma dessas estratégias tem o objetivo de melhorar a população inicial da busca em termos de diversidade e adequação para otimizar o objetivo. Eles usam um conjunto de programas em Java para avaliar essas estratégias de inicialização da população inicial e mostram que elas podem produzir resultados muito diferentes. Estratégias desse tipo foram implementadas na ferramenta EVOSUITE, desenvolvida pelos autores (Gordon e Arcuri, 2012). Eles afirmam que quando se tem algum conhecimento do domínio do problema, torna-se mais fácil produzir uma população inicial.

Vários autores discutem de forma mais genérica como gerar casos de teste iniciais em algoritmos de teste baseado em busca, como por exemplo Yang et al. (2006). Também nessa linha, McMinin et al. (2012) investigaram estratégias para criar a população inicial para algoritmos de geração de casos de teste especificamente para dados do tipo *string*.

Kukolj et al. (2013) combinam técnicas de teste estrutural e funcional para reduzir o conjunto de casos de teste. Na abordagem proposta, os dados de cobertura são utilizados para selecionar os casos de teste apropriadamente e ordená-los em uma ordem requerida, baseada em uma dada especificação, isto é, critérios de cobertura. Embora esse conjunto não seja utilizado como população inicial em algoritmos de busca, a abordagem torna possível obter a mesma cobertura, mas com menos casos de teste.

3.4.2 Outros trabalhos

Doganay et al. (2014) apresentam um estudo de caso de uma aplicação de técnicas de teste de software baseadas em busca para um software embarcado industrial no domínio de telecomunicações, com o objetivo de gerar dados de entrada no nível de teste de unidade, considerando os critérios de cobertura todos-nós e todas-arestas do grafo de fluxo de controle. Com o estudo de caso conduzido verificou-se que a abordagem aumentou a cobertura mas, no entanto, a geração foi tão efetiva quanto a geração aleatória.

Micskei et al. (2012) propõem uma abordagem de geração de dados de teste para teste baseado em modelos e utilizando técnicas baseadas em busca. Os dados gerados representam situações complexas e são utilizados para avaliar as trilhas (*traces*) de teste e para computar métricas de cobertura. Ela se baseia na modelagem do contexto em que são incluídas as restrições e condições que determinam as situações normais e excepcionais e os métodos definidos para gerar sistematicamente os testes de estresse. No algoritmo de busca, as soluções são geradas como instâncias de um modelo conforme o contexto, as

instâncias são manipuladas realizando transformações no modelo como, adicionando e removendo elementos do contexto e fragmentos de/para os candidatos a solução. As funções de *fitness* são baseadas nos requisitos do contexto e nas estratégias de teste que incluem: criação de um número mínimo de objetos de contexto que são necessários para testar a satisfação de um requisito; combinação de modelos de contexto relacionados a diferentes requisitos para testar onde os contextos combinados irão violar esses requisitos; criação de modelos de contexto que estão no limite ou violam sistematicamente as restrições de domínio-específico selecionadas para o teste de robustez. Uma vez que o dado de teste gerado é executado, a satisfação dos requisitos é verificada e a cobertura com respeito aos cenários de contexto é calculada.

Braione et al. (2014) realizaram um estudo empírico que confronta um conjunto de geradores de dados de teste em estado de pesquisa com quatro versões de um programa de um sistema controlador em tempo real e de segurança crítica. O programa foi desenvolvido em C, implementa várias computações não lineares e é integrado dentro de uma tarefa de controle dependente de tempo. Além disso, possui algumas características desafiadoras como estruturas inter-procedurais, aritmética de ponto flutuante e asserções de código a serem utilizadas como oráculos de teste indisponíveis. O experimento considerou geradores de teste baseados em teste aleatório, execução simbólica dinâmica e baseados em busca que foram desenvolvidos para geração de dados de teste de softwares tradicionais. Como resultado principal, o estudo contribui com evidência empírica de que os geradores de teste podem ser eficazes em um software industrial, expondo falhas que não foram detectados durante o teste de implantação de um protótipo de um sistema de segurança crítica. Os defeitos encontrados neste estudo dizem respeito a questões de robustez desconhecidas com pré-condições implícitas não verificadas e possível estouro de pontos flutuantes, comportamentos de casos extremos nas entradas singulares, e inconsistências indesejadas entre a re-implementação de um algoritmo principal e suas versões de base.

He et al. (2011) propõem um algoritmo para geração de casos de teste baseado em mutação para modelos Simulink que combina teste estrutural com análise formal de conceitos. A abordagem é capaz de gerar pequenos conjuntos de casos de teste que alcançam alta cobertura em uma coleção de modelos Simulink a partir do domínio automotivo. Foram conduzidos experimentos que mostraram que o algoritmo proposto executa melhor que o teste aleatório ou abordagens de teste de mutação mais simples.

3.5 Considerações finais

Neste capítulo foram apresentados e discutidos os principais trabalhos referentes ao teste de software e geração de dados de teste em sistemas embarcados com enfoque no teste

Tabela 3.5: Resumo dos principais trabalhos referentes ao teste estrutural de software em sistemas embarcados

Trabalho	Define técnica ou critério	Realiza experimentos	Possui ferramenta
Li et al. (2008)	Não	Estudo de caso	Não
Yang et al. (2006)	Não	–	–
Wu et al. (2007)	Não	Estudo de caso	exVantage
Guan et al. (2006)	Critérios CACC, mas não são exclusivos para sist. emb.	Estudo de caso	–
Gross et al. (2009)	Não	Estudo de caso	EvoTest
Jürjens et al. (2008)	Não	Estudo de caso	Test Real Time

Tabela 3.6: Resumo dos principais trabalhos referentes à geração de dados de teste em sistemas embarcados

Trabalho	Técnica utilizada	Possui ferramenta
Wegener et al. (1997)	algoritmos genéticos e geração aleatória	Não
Nilsson e Henriksson (2005)	técnica baseada em mutação	extensão da ferramenta TrueTime
Tlili et al. (2006)	algoritmo evolutivo	Não
Windisch (2010)	algoritmo evolutivo	Sim

estrutural encontrados na literatura. Também foram vistas ferramentas que dão apoio ao teste estrutural para esses tipos de sistemas. As Tabelas 3.5 e 3.6 apresentam um resumo dos principais trabalhos referentes ao teste estrutural de software e geração de dados de teste, respectivamente, em sistemas embarcados.

De acordo com a revisão sistemática conduzida, pode-se notar que há poucos trabalhos preocupados com o teste estrutural e geração de dados de teste específicos para sistemas embarcados. Os critérios e técnicas utilizados até então são os mesmos que os definidos para os softwares tradicionais; a maioria das abordagens para geração de dados de teste preocupa-se apenas com uma das características de sistema embarcado como, por exemplo, tempo real. Além disso, a maior parte dos trabalhos trata de estudos de casos para uma aplicação específica em que se tenta verificar como realizar o teste considerando as restrições/dificuldades do sistema em estudo. Em relação aos trabalhos de geração automática de dados de teste observa-se que a maioria deles considera variáveis de entradas mais simples e comuns, como números e vetores. No entanto, em domínios como os veículos autônomos há a ocorrência de tipos mais complexos de dados. Nota-se, entretanto, que é uma área de pesquisa promissora e que seria interessante que fossem investigadas abordagens e estratégias que permitam a utilização do teste estrutural em sistemas embarcados, bem como mecanismos de apoio automatizado às aplicações dessas abordagens.

Veículos autônomos

4.1 Considerações iniciais

Os veículos autônomos podem trazer melhoras no trânsito como, por exemplo, aumento da capacidade da estrada e do fluxo de tráfego por causa do rápido tempo de resposta, menos consumo de combustível e poluição graças a uma condução mais cautelosa, poucos acidentes graças aos sistemas de prevenção de colisão. No entanto, há alguns desafios que precisam ser considerados como navegação em terrenos desertos, planos e montanhosos; lidar com obstáculos como pontes, passagem de pedestres, buracos e outros veículos; obedecer regras de trânsito; entrar de forma segura no fluxo de tráfego e passar por cruzamentos movimentados; seguir e ultrapassar veículos em movimento; dirigir em uma rota alternativa quando a primeira rota está bloqueada; estacionar de forma correta; e a regra mais importante: sem colisões.

Este capítulo tem como objetivo apresentar as características dos veículos autônomos inteligentes. Para isso, na Seção 4.2 os veículos autônomos são apresentados com mais detalhes; na Seção 4.3 é apresentado um exemplo de um sistema controlador de um veículo autônomo para navegação e desvio de obstáculos. Esse exemplo será utilizado para validar as propostas apresentadas nos próximos capítulos desta tese. A Seção 4.4 descreve como é o processo do teste do software de alguns veículos autônomos e, por fim, na Seção 4.5 são feitas algumas considerações finais.

4.2 Veículos autônomos inteligentes

Veículos autônomos inteligentes (ou apenas veículos autônomos – VA) são um tipo de Robô Móvel Autônomo (RMA) que, por sua vez, é uma combinação de vários componentes físicos (hardware) e computacionais (software) e têm como principal característica a capacidade de locomoção e de operação de modo semi ou completamente autônomo. Um robô alcança um maior nível de autonomia à medida que passa a integrar características como capacidade de percepção (sensores que são capazes de “ler” o ambiente em que ele atua), capacidade de ação (atuadores e motores capazes de produzir ações como, por exemplo, o deslocamento do robô no ambiente), robustez e inteligência (o robô é capaz de resolver e executar tarefas lidando com as mais diversas situações) (Wolf et al., 2009).

Os RMAs possuem diferentes configurações de dispositivos de hardware embarcados, de acordo com as funções e as tarefas para as quais são projetados. Os principais dispositivos de hardware de um robô são seus sensores e atuadores. Há vários tipos de sensores como de posição e de orientação (GPS, Bússola), de obstáculos (sensor infra-vermelho), de contato, deslocamento, velocidade. Rodas, esteiras, juntas e articulações são exemplos de elementos físicos que podem ser acionados por atuadores. A integração das informações vindas de todos os sensores, com o objetivo de gerar comandos e controlar os diferentes dispositivos de atuação do robô, juntamente com a especificação e seleção de diferentes componentes mostra quão complexo o projeto de um sistema robótico pode ser. Além disso, deve-se garantir que a tarefa seja executada de modo correto e sem colocar em risco tanto o robô quanto aqueles que o cercam. Sendo assim, seu desenvolvimento deve envolver um cuidadoso projeto tanto de hardware como de software, garantindo um controle robusto, tolerante a falhas e a situações imprevistas, constituindo, então, um “sistema de controle inteligente” (Jung et al., 2005; Wolf et al., 2009).

Os sistemas de apoio à condução de veículos são uma importante aplicação de robótica móvel, seja de modo completamente autônomo ou de modo semi-autônomo, atuando para aumentar a segurança do condutor e das vias em geral. Nesse sentido, diversas soluções eletrônicas e de informática vêm sendo desenvolvidas e implementadas nos veículos, como freios ABS e sistemas de navegação (Jung et al., 2005; Wolf et al., 2009). A tecnologia *drive-by-wire* representa uma nova era, na qual soluções mecânicas são substituídas por soluções eletromecânicas. A possibilidade de dirigir um veículo a partir de sinais eletrônicos permite o uso de computadores embarcados atuando como copilotos e ajudando o motorista em situações de emergência, ou na execução de tarefas, tais como estacionar um veículo.

A estrutura de um veículo terrestre inteligente com tecnologia *drive-by-wire* é apresentada na Figura 4.1. Ela consiste em uma unidade de controle estruturada em diferentes

níveis hierárquicos de controle. Os sistemas mecatrônicos embarcados baseiam-se em informações internas e externas ao veículo obtidas por meio de sensores específicos. Um sistema de comunicação complementa a integração de toda a estrutura de controle, possibilitando a troca de informações entre o veículo e uma base de supervisão e comando remotos, bem como a comunicação entre veículos. Na parte de sensoriamento há três grupos básicos de sensores: Sensores para Reconhecimento de Rota, Sensores para Reconhecimento de Objetos & Obstáculos e Sensores de Navegação, que permitem que a rota traçada seja seguida com segurança (Jung et al., 2005).

Figura 4.1: Estrutura de um veículo inteligente (retirado de Jung et al. (2005))

A fim de tornar a experiência de direção mais segura e conveniente, por décadas, pesquisadores de diferentes áreas da robótica, engenharia de automação e áreas relacionadas têm tentado desenvolver sistemas inteligentes para veículos modernos. Esses sistemas são projetados para ajudar a dirigir automaticamente ou monitorar um condutor humano e assisti-lo na navegação. Eles podem avisar o condutor no caso de uma situação perigosa e fornecer meios de evitar colisões ou mitigar as consequências se houver uma colisão inevitável. Além disso, esses veículos inteligentes devem ser capazes de operar em todas as situações de tráfego, seja em rodovias ou em vias urbanas lotadas.

De alguma forma, hoje em dia, esses avanços dos sistemas inteligentes para veículos têm se tornado possível graças às mais recentes tecnologias automotivas incluindo vários sensores, atuadores, processadores, componentes de comunicação e algoritmos avançados. Além disso, o custo dos sensores e processadores vêm caindo rapidamente combinado com o aumento dos poderes dos sensores em fornecer a base para o crescimento contínuo da inteligência do veículo.

Segundo Jo et al. (2014), um veículo autônomo possui cinco funções básicas: percepção, localização, planejamento, controle e gerenciamento do sistema. A Figura 4.2 apresenta uma descrição conceitual de cada uma dessas funções. Percepção é o processo de como o robô irá perceber suas propriedades e seu ambiente utilizando vários tipos de técnicas

de sensoriamento como LIDAR e visão computacional. A localização encontra a posição do veículo utilizando técnicas de um Sistema de Posicionamento Global (GPS), navegação estimada (*dead reckoning*) e mapas de estrada. A função de planejamento determina o comportamento e o movimento do veículo autônomo baseando-se nas informações de percepção e localização. A função de controle segue os comandos desejados a partir da função de planejamento, dirigindo, acelerando e freando o veículo autônomo. Finalmente, o sistema de gerenciamento supervisiona o sistema global de condução autônoma. Alguns exemplos de funções do gerenciamento do sistema são o sistema de gerenciamento de falhas, sistema de *logging* e interface humano-máquina. Quase todos os veículos autônomos possuem essas cinco funções básicas e cada uma dessas funções têm diferentes subcomponentes funcionais de acordo com o propósito e complexidade do veículo autônomo (Jo et al., 2014).

Figura 4.2: Funcionamento básico de um veículo autônomo (retirado de Jo et al. (2014))

4.2.1 Sensores e atuadores

Para que um veículo autônomo consiga obter informações a respeito de si mesmo e de seu ambiente é necessário utilizar dispositivos, chamados de sensores, que realizam medições e extraem informações significativas a partir dessas medições (Romero et al., 2014; Siegwart e Nourbakhsh, 2004). Há vários tipos de sensores como os **odômetros óticos**, que servem para controlar a posição e a velocidade das rodas do robô, fornecendo uma estimativa da distância relativa navegada; os **sensores de ultrassom ou sonares**, que têm como princípio básico a velocidade de propagação do som no ar; os **sensores a laser**, chamados

no meio acadêmico de LIDAR (*Light Detection And Ranging*) e que permite detectar objetos remotamente, usando para isso *laser* pulsado; **sensores de posição, orientação e movimentação** como a bússola, o GPS, os acelerômetros e giroscópios. Atualmente pode-se encontrar no mercado dispositivos integrados, as IMUs (*Inertial Measurement Units*), que geralmente integram os dados de mais de um tipo de sensor (GPS, bússola, acelerômetro e giroscópio) visando fornecer uma informação mais confiável da localização e orientação do robô (Romero et al., 2014).

Sensores do tipo LIDAR capturam nuvens de pontos que são utilizadas por algoritmos de percepção. Uma nuvem de pontos é uma coleção de nD pontos em que, geralmente, $n = 3$. Ela é utilizada para representar informações 3D sobre o mundo. As coordenadas x , y e z normalmente representam, respectivamente, a distância em que o ponto está do carro, a posição horizontal em relação ao carro e a altura do ponto. Além das informações das coordenadas xyz, cada ponto p também pode conter outras informações adicionais como, por exemplo, cores RGB, valores de intensidade, distância e resultados de segmentação. Portanto, um ponto p é representado como uma tupla $p_i = \langle x_i, y_i, z_i, r_i, g_i, b_i, dist_i, \dots \rangle$ e uma nuvem de pontos é representada como uma coleção de pontos $P = \{p_1, p_2, \dots, p_i, \dots, p_n\}$. Uma nuvem de pontos é geralmente representada por uma matriz $NP(N, M)$ contendo $N \times M = nD$ pontos. Portanto uma célula da matriz $NP(i, j)$ contém um ponto $p_{i,j}$ com $i = 1, 2, \dots, N$ e $j = 1, 2, \dots, M$.

Além de perceber o ambiente, os veículos autônomos devem atuar sobre ele, ou seja, devem poder se deslocar, manipular e interagir de alguma maneira com os elementos em seu ambiente. Para que isso seja possível, os veículos autônomos devem conter mecanismos de atuação como motores, volantes, freios, ou controles de aceleração, que são capazes de produzir as ações requeridas pelo veículo para que ele possa desempenhar uma determinada tarefa (Romero et al., 2014).

4.2.2 Percepção

A habilidade de perceber o ambiente local do veículo é um dos principais desafios no campo de veículos autônomos terrestres. Condições de ambiente como luz ou cores mudam frequentemente e há também vários objetos dinâmicos e estáticos no cenário a ser levado em conta. Os melhores resultados de percepção são tipicamente alcançados aproveitando-se dos pontos fortes dos sensores mencionados anteriormente. Algumas percepções importantes no campo de veículos autônomos e suas realizações são: estimação do estado do veículo e compensação do próprio movimento; objetos estáticos; participantes do tráfego e outros veículos em movimento; estimação do formato da estrada e localização auxiliada por mapa (Luettel et al., 2012).

4.2.3 Comportamento, navegação e controle

Cenários urbanos e de estrada expõem uma variedade de situações de navegação que requerem um tratamento individual, adaptados a uma determinada situação. Navegação ao longo de uma pista improvisada em uma floresta com um GPS em condições difíceis pode requerer um tratamento diferente do que uma navegação ao longo de um terreno aberto. O mesmo também vale para a navegação em uma zona de estacionamento versus a direção em rodovias. Para a coordenação de comportamento em diferentes situações, máquinas de estados finitos e hierárquicas são ferramentas comuns para que o veículo possa decidir o que fazer, mas as maiores diferenças estão nas abordagens de navegação (Luettel et al., 2012).

4.2.4 Exemplos de veículos inteligentes

Um fator que gerou grande interesse no desenvolvimento de veículos inteligentes foi a competição “*Grand Challenge*” promovida pela DARPA (*Defense Advanced Research Projects Agency*). Essa competição foi destinada ao desenvolvimento de veículos autônomos que precisavam completar uma série de objetivos pré-estabelecidos.

O DARPA “*Grand Challenge*” teve início em 2004 e o objetivo era desenvolver veículos terrestres capazes de navegar em trilhas no deserto e em estradas com alta velocidade de maneira completamente autônoma. A equipe do primeiro veículo que completasse o trajeto receberia um prêmio de um milhão de dólares. Entretanto, o veículo que foi mais longe não percorreu mais do que 14km (o percurso total era de 300km), mesmo tendo custado o triplo do valor do prêmio. A competição se repetiu em 2005 e o veículo vencedor do desafio foi o Stanley da equipe de Stanford (Thrun et al., 2006).

Em 2007, o objetivo foi alterado para que os veículos autônomos dirigissem por 97km em um ambiente urbano, interagindo com outros veículos em movimento e obedecendo às leis de trânsito local. Esse desafio ficou conhecido como “*Urban Challenge*” e o veículo vencedor foi o Boss, da Carnegie Mellon University (Urmson et al., 2008).

O principal desafio tecnológico no desenvolvimento do veículo Stanley era construir um sistema altamente confiável, capaz de dirigir em velocidades relativamente altas por diversos ambientes e estradas não estruturados, e fazer tudo isso com alta precisão. Esses requisitos levaram a uma série de avanços na área de navegação autônoma. Métodos em áreas como percepção de terreno de longo alcance e prevenção de colisões em tempo real foram desenvolvidos ou estendidos (Thrun et al., 2006).

O veículo Boss é composto de subsistemas de planejamento de movimento, de percepção, de planejamento de missão e de comportamento. O subsistema de planejamento de movimento é capaz de evitar obstáculos estáticos e dinâmicos. Esse subsistema tam-

bém considera o cenário de direção estruturada (seguindo uma estrada) e não estruturada (manobras em estacionamento). O subsistema de percepção processa e combina dados de múltiplos sensores do Boss para fornecer um modelo de composição do mundo para o resto do sistema. O modelo consiste em três partes principais: um mapa de obstáculo estático, uma lista dos veículos em movimento no mundo e a localização do Boss em relação à estrada.

O planejador de missão calcula o custo de todas as rotas possíveis para o ponto próximo à missão dada, tendo como conhecimento a rede rodoviária; ele planeja uma rota a partir de sua posição atual até um destino. O sistema de comportamento formula uma definição do problema para o planejador de movimento, que por sua vez, deve resolver esse problema com base nas informações estratégicas fornecidas pelo planejador de missão; ele faz as decisões táticas para executar o plano de missão e lida com a recuperação de erro quando há problemas (Urmson et al., 2008).

O Boss também possui uma infraestrutura de software e ferramentas que habilitam os outros subsistemas e fornecem uma base sobre a qual os algoritmos são implementados. Também fornece ferramentas para que se possa realizar o log dos dados online, reproduzir esse log de dados de modo *offline* e ferramentas de visualização que auxiliam os desenvolvedores na construção e resolução de problemas do sistema (Urmson et al., 2008).

O primeiro veículo a navegar de forma autônoma pelas ruas na América Latina foi o CaRINA II (Fernandes et al., 2014), desenvolvido pelo Laboratório de Robótica Móvel do ICMC/USP. O projeto CaRINA iniciou com um carro elétrico modificado para ser controlado por computador e instrumentado com vários sensores de percepção. Mais tarde, um Fiat Palio Adventure padrão foi modificado para atuação computacional da direção, aceleração e freio e recebeu o nome de CaRINA II. Para o desenvolvimento tanto do CaRINA I quanto do CaRINA II, o ROS (*Robot Operating System*) (ver Seção 4.2.6) foi escolhido como o pacote de ferramentas principal.

4.2.5 Exemplos de arquiteturas de veículos inteligentes

A Figura 4.3 apresenta a arquitetura do sistema do CaRINA. Módulos específicos para cada acionamento do motor do veículo foram desenvolvidos e são responsáveis por converter os comandos abstratos (*set points*) em seus valores equivalentes específicos para cada plataforma, produzindo a ação desejada. Esses módulos são responsáveis pela comunicação com dispositivos de controle de cada sistema e encapsulam protocolos de baixo-nível, atuando como um *driver* de software para os dispositivos do veículo. Além disso, esses módulos são responsáveis por manter a comunicação dos canais ativa e garantir a integridade da informação transferida. Os módulos de tele operação e controle de velocidade interagem diretamente com eles. O módulo de tele operação permite que operadores hu-

manos controlem cada um desses três sistemas utilizando um *joystick*, cooperativamente (soma dos comportamentos) ou selecionando para sobrescrever (ignorar) os comandos diretamente aos dispositivos do veículo, ignorando outros módulos e comandos.

Baseada na arquitetura proposta, os pesquisadores desenvolveram quatro sistemas de navegação. Um desses sistemas utiliza visão-estéreo, conforme apresentado com mais detalhes na Seção 4.3

Figura 4.3: Arquitetura de sistema do CaRINA (Fernandes et al., 2014)

A Figura 4.4 apresenta a arquitetura de software do Stanley, que consiste de cerca de 30 módulos executados em paralelo. O sistema é dividido em seis camadas que correspondem às seguintes funções: interface de sensor (que compreende uma série de módulos de software preocupados com o recebimento e o *timestamp* de todos os dados do sensor), percepção (mapeia os dados do sensor em modelos internos), controle (responsável por regular o volante, acelerador e freio, dependendo da resposta do veículo), interface do veículo (serve como uma interface para o sistema *drive-by-wire* do robô), interface do usuário (compreende módulos de parada de emergência remota e de início do software) e serviços globais (fornece vários serviços básicos para todos os módulos de software) (Thrun et al., 2006).

4.2.6 Robot Operating System (ROS)

O ROS tem-se tornado uma das plataformas mais importantes para pesquisa e desenvolvimento em robótica (Fernandes et al., 2014). Ele oferece funcionalidades similares às de um sistema operacional. Foi desenvolvido originalmente pelo Stanford Artificial Intelligent Lab e mais tarde no Willow Garage.

Figura 4.4: Fluxograma da arquitetura de software do Stanley (retirado de Thrun et al. (2006))

ROS utiliza o conceito de **nós** que, na verdade, são processos do sistema operacional que executam uma computação. Esses nós podem residir em diferentes máquinas de maneira transparente. Os nós tomam conhecimento um dos outros por meio do processo *roscore*, que atua principalmente como um servidor de nomes. A Figura 4.5 apresenta um exemplo de nós do ROS e o roscore (ROS.org, 2015).

Figura 4.5: Exemplo de nós do ROS

Os nós se comunicam uns com os outros por meio de mensagens. Para enviar mensagens de um nó para um ou mais nós, o ROS utiliza o mecanismo de tópicos, que segue o padrão *publisher-subscriber*, conforme apresentado na Figura 4.6. Nós publicam (escrevem) e/ou subscrevem (leem) a partir de um tópico que pode ser local ou remoto. Pode haver vários *publishers* e *subscribers* concorrentes para um único tópico e um único tópico pode publicar (*publisher*) e/ ou se inscrever (*subscriber*) em múltiplos tópicos. Os *publishers* e *subscribers* não estão cientes da existência uns dos outros.

Figura 4.6: Comunicação entre os nós no ROS

O ROS é baseado em ferramentas, isto é, um grande número de ferramentas pequenas e específicas são usadas para criar e executar vários componentes. Entre essas ferramentas estão as responsáveis por criar, armazenar, processar, analisar, visualizar e reproduzir todas as mensagens recebidas pelo veículo durante o teste de campo (ver Seção 4.4). Ele também é totalmente integrado com bibliotecas importantes, como: OpenCV, que fornece uma extensiva biblioteca de funções, e métodos para Visão Computacional e Processamento de Imagem; PCL, que permite o processamento e visualização de nuvens de pontos 3D (por exemplo, dados de laser e câmera estéreo); Gazebo que é uma ferramenta de simulação virtual realística (VST); e Player, que fornece uma interface de rede para uma variedade de hardware de robôs e sensores. Como um sistema meta-operacional para robôs ele fornece: abstração de hardware, controle de dispositivos de baixo nível, funcionalidades comumente utilizadas, troca de mensagens entre processos e gerenciamento de pacotes.

4.3 Exemplo do sistema de navegação e desvio de obstáculos do veículo CaRINA I

Como exemplo de um programa controlador de veículo autônomo tem-se o módulo do sistema de navegação e desvio de obstáculos desenvolvido para o veículo CaRINA I e que é utilizado para validar as propostas apresentadas nesta tese. O objetivo desse módulo de software é dirigir um veículo autônomo em uma região urbana para chegar a um determinado ponto definido por uma coordenada de GPS, evitando colisões. Ele é dotado de uma câmera estéreo que fornece duas imagens, as quais são processadas por um método semi-estéreo global para produzir um mapa de disparidade. Esse mapa é então convertido para uma nuvem de pontos em 3D com base em parâmetros da câmera. A orientação da câmera em relação ao solo é estimada pelo método RANSAC (*RANdon SAMple Consensus*) e é usado um método de detecção de obstáculos que classifica os pontos com base em elevações e diferenças relativas de altura. Essas informações são usadas como entradas para o *Vector Field Histogram* (VFH) que é usado para guiar o veículo até o ponto de chegada (Fernandes et al., 2012; Mendes, 2012; Mendes e Wolf, 2013). O diagrama de classes desse programa é mostrado na Figura 4.7.

Figura 4.7: Diagrama de classes

O fluxo de execução do programa de controle do veículo pode ser descrito simplificada-mente pelo grafo de chamadas mostrado na Figura 4.8. A partir do método `main()` há um laço principal que recebe a leitura da câmera, faz os processamentos descritos no parágrafo acima e toma a decisão sobre a atuação do veículo: acelerar, desacelerar, freiar, mudar o ângulo das rodas etc. Os dois métodos principais do programa, responsáveis

Figura 4.8: Grafo de chamadas

por esses cálculos e decisões são *process()*, da classe *cObstacleAvoidance*, e *calcule()*, da classe *cVFH*. O primeiro tem 155 linhas de código sem comentários e o segundo 153.

O método *process()* recebe a nuvem de pontos como entrada e a passa como parâmetro para os métodos *calculatePlane()* e *rotatePlaneToZ()* da classe *cFindPath* a fim de filtrá-la, segmentando o conjunto de pontos para calcular o plano. Em seguida, utiliza o método *detect()* da classe *cObstacleDetection* para determinar se a área é ou não navegável ou se não se sabe e, com as informações recebidas da variável de entrada *pose*, o método *getDirection()* da classe *cGPSNavigation* calcula o ângulo de distância até o objetivo. O método *checkGoal()*, da mesma classe, verifica se já atingiu o objetivo. O método *calcule* da classe *cVFH* é responsável pelo desvio de obstáculos, se houver. Ele modifica a variável *state*, utilizada no método *process()*, para um dos valores: *ToGoal*, *OnObstacle* e *Change*.

4.4 Teste no contexto processo de desenvolvimento de robôs móveis autônomos

Na Seção 4.2.4 são discutidos dois exemplos de veículos autônomos reais: Stanley (Thrun et al., 2006) e Boss (Urmson et al., 2008, 2006) que competiram em eventos para avaliar a capacidade de carros autônomos trafegarem em ambientes urbanos e rurais de forma autônoma. Na mesma linha Suganuma e Uozumi (2012) apresentam um veículo que também participou de um evento demonstrativo de veículos autônomos. Esses artigos usualmente descrevem a arquitetura do software controlador do veículo e aspectos do processo de desenvolvimento, incluindo teste.

Urmson et al. (2008) são os autores que melhor descrevem o processo de desenvolvimento de software para veículos autônomos usado para o desenvolvimento do Boss. Para eles, o teste e o desenvolvimento estão intimamente interligados e seguem um processo cíclico, conforme apresentado na Figura 4.9.

Eles descrevem o processo, resumidamente, da seguinte forma: antes do desenvolvimento de algoritmos começar, a equipe reúne os requisitos que definem as funcionalidades que o Boss deve implementar para ser capaz de completar o desafio. Algoritmos e abordagens são testados de maneira *offline*, seja por simulação ou reproduzindo dados previamente gravados. Uma vez que um algoritmo torna-se suficientemente maduro, ele é embarcado no veículo de teste, a fim de avaliar as interações do sistema com o ambiente. Esses testes, muitas vezes, revelam problemas de algoritmos ou erros de implementação que não foram percebidos durante o teste *offline*, resultando em numerosos ciclos de retrabalho e depuração. Uma vez que os desenvolvedores estão satisfeitos com o funcionamento do algoritmo, eles são adicionados aos testes de sistema (que inclui o carro, o hardware e o software) formal e regularmente agendados. Testes independentes dos algoritmos, muitas vezes, descobrem novas deficiências, exigindo retrabalho. Em outros casos, testes e **pós-análise** (ou pós-processamento) fazem com que a equipe modifique os requisitos que impulsionaram o desenvolvimento, limitando ou estendendo o escopo, conforme apropriado. Após algum tempo, a tecnologia é considerada aceita e pronta para participar do desafio. Durante os testes de regressão, a equipe avalia o desempenho do carro com relação a um conjunto de cenários descritos em uma cartilha. Cada cenário é anotado com uma prioridade (classificado de 1 a 3), como deve ser testado, como se relaciona com os requisitos e uma descrição de como o carro deve se comportar quando encontrar esse cenário.

Figura 4.9: Processo de desenvolvimento utilizado no desenvolvimento do Boss (Urmson et al., 2008)

Não é difícil perceber que há três momentos em que ocorre teste nesse processo de desenvolvimento. No primeiro ocorre o teste dos componentes (algoritmos) individualmente e depois eles são integrados e testados. Ambientes de simulação são usados para apoiar o

teste neste momento. No segundo momento os componentes integrados são embarcados em um veículo e o conjunto é avaliado como um sistema. Esse é um teste essencialmente funcional e baseado em cenários. Durante esses testes podem ser coletados dados que em um terceiro momento podem ser usados para reexecutar o software (ou parte dele) sob diferentes cenários ou situações (teste de regressão).

O segundo momento, ou segunda fase da descrição de Urmson et al. (2008) é chamado por Trepagnier et al. (2006) de “teste de campo”. Eles dizem que no teste de campo o “o veículo passa por testes árduos projetados não apenas para testar a confiabilidade e resistência do software e do hardware, mas também o desempenho do veículo... passando por terrenos difíceis, cenários difíceis de obstáculos a serem evitados e longos tempos de execução”. Os testes de campo não podem ser refeitos exatamente com os mesmos dados de entradas, pois eles são coletados *online* durante a execução do teste, mas os dados de entrada podem ser coletados e usados posteriormente em execuções simuladas. Crane et al. (2007) também usam o termo teste de campo para se referirem a esse tipo de teste.

Tanto Bacha et al. (2008b) como Thrun et al. (2006) e Rauskolb et al. (2008) enfatizam a importância dos testes de regressão usando dados coletados em testes de campo ou testes baseados em simulações. Eles comentam sucintamente sobre a existência de ferramentas de visualização no ambiente de desenvolvimento, bem como para reexecução do software a partir de dados coletados e armazenados em arquivos de *log*, sem apresentar qualquer exemplo ou detalhe.

Os testes de campos são os testes finais e decisivos para assegurar que o veículo autônomo se comporte como esperado, mas os autores mencionados neste capítulo não levam em consideração nenhuma informação a respeito da estrutura do código fonte durante os testes, isto é, não fazem referência a esse tipo de teste. É possível que um veículo tenha realizado vários testes de campo com sucesso mas importantes partes do código fonte nunca tenham sido executadas.

A exceção são os autores do veículo desenvolvido por Rauskolb et al. (2008), que usam um processo de desenvolvimento em que a cobertura estrutural é analisada. Segundo eles, o processo de teste utilizado cobre apenas a fase de teste de unidades para algumas funções ou partes do sistema de software. A cobertura analisada é baseada na contagem dos comandos executados.

4.5 Considerações finais

Neste capítulo foram apresentadas as características principais de veículos autônomos e discutidos os processos de desenvolvimento e teste do software que controla o veículo,

com ênfase em testes estruturais. Foram apresentados três exemplos reais de veículos inteligentes e discutidas as suas características principais.

Considerando o que foi apresentado, nota-se que há três características importantes em relação à criação dos dados de entrada de testes de campo de veículos autônomos que dificultam o teste. A primeira é que os dados de entradas devem ser numerosos para fazerem sentido e isso dificulta a sua preparação manual antes de que o programa esteja pronto. Por exemplo, milhares de leitura de dados são realizadas com elevada frequência durante o movimento do veículo. Por isso é geralmente usada a estratégia de realizar um primeiro teste de campo e coletar os dados de entrada provenientes de câmeras e sensores. A segunda característica é a dificuldade de garantir que os dados de entrada sejam repetidos, até mesmo quando o teste é refeito reutilizando cuidadosamente um mesmo cenário, como por exemplo, partir da mesma posição inicial e atingir a mesma posição final. A terceira dificuldade é que os dados de entradas são estruturas complexas. Por exemplo uma nuvem de pontos é representada por uma matriz de grande dimensão, com cada célula contendo diversas informações.

A partir desta revisão da literatura da área e também de contatos com alguns grupos de pesquisadores de veículos autônomos brasileiros, observou-se que geralmente o teste estrutural e análises de cobertura são pouco usados nos processos de desenvolvimento utilizados. Uma das justificativas para a falta de ênfase dada ao teste estrutural de software para veículos autônomos é, além das dificuldades mencionadas no parágrafo anterior, que ele depende fortemente de ferramentas de software disponíveis no ambiente de desenvolvimento e na dinamicidade do teste. De acordo com o mapeamento sistemático descrito no Capítulo 3, há poucas abordagens e ferramentas que levam em conta o teste estrutural em sistemas embarcados, principalmente no contexto de veículos autônomos. Dentre as ferramentas existentes, não encontrou-se nenhuma que considerasse estruturas de dados complexas em sua abordagem. Em geral, os estudos de caso conduzidos com as ferramentas existentes para análise de cobertura e geração de dados de teste consideram ou programas tradicionais ou programas que contém variáveis de entrada com tipos de dados simples.

Assim, nos próximos capítulos são apresentadas propostas de modelos, algoritmos, heurísticas, estratégias e ferramentas para apoiar o teste de campo de sistemas que possuem as características resumidas acima.

Ambiente de apoio ao teste de campo de veículos autônomos

5.1 Considerações iniciais

A partir da revisão de literatura conduzida nesta tese, pode-se observar que o teste estrutural de software é pouco utilizado no processo de desenvolvimento de veículos autônomos e uma das razões para isso é a falta de abordagens e ferramentas que levam em conta as características inerentes desses sistemas. Nesse contexto, esta tese propõe uma abordagem que inclui um modelo conceitual, uma ferramenta que implementa esse modelo, heurísticas, algoritmos e estratégias para oferecer suporte ao teste de campo de veículos autônomos.

Este capítulo apresenta um modelo conceitual para um ambiente de apoio ao teste de campo com base no teste estrutural e no qual é baseada uma ferramenta de software que foi desenvolvida pela autora da tese. Tomou-se a decisão de apresentar todo o modelo conceitual desenvolvido e descrever seus componentes, porque a própria ferramenta foi usada para gerar os resultados que são mostrados neste e nos próximos capítulos. Portanto, este capítulo apresenta o modelo proposto com foco para pós-análise do teste de campo, reproduzindo o programa com os logs coletados em campo bem como a implementação dessa parte da ferramenta.

Este capítulo é organizado como segue: a Seção 5.2 apresenta o modelo proposto e descreve os seus principais componentes; a Seção 5.3 apresenta algumas informações sobre o projeto da ferramenta desenvolvida e sua arquitetura; a Seção 5.4 apresenta sucintamente o uso da ferramenta para pós-análise do teste de campo, sua interface e alguns resultados e, por fim, a Seção 5.5 apresenta as conclusões deste capítulo.

5.2 Um modelo para apoiar o teste estrutural de veículos autônomos

Conforme discutido na Seção 4.4, uma prática útil durante o teste de veículos autônomos é armazenar os dados de entrada a partir de um teste de campo para apoiar análises *offline* mais aprofundadas, tal como mostrado na Figura 5.1. O desenvolvimento de software (e de sistemas) geralmente segue numerosas interações em que os dados de entrada dos testes de campo com base em cenários são armazenados e, posteriormente, são reexecutados ou reproduzidos *offline* com a ajuda de ferramentas de depuração. Eles também podem ser modificados, ou outros logs podem ser gerados com base neles para servirem como entradas em simulações do programa, com vistas a aumentar as coberturas obtidas. As ferramentas existentes no ambiente fornecem diferentes visões do programa e permitem inspeções de dados e dos valores das variáveis (Thrun et al., 2006).

Figura 5.1: Esquema para gerar dados de teste

Compreender e analisar as instruções ou outros elementos estruturais do programa que não tenham sido exercitados (cobertos) em um cenário de teste pode ser útil para avaliar a qualidade do teste. Além disso, o teste estrutural também pode contribuir para o planejamento e análise de cenários de teste. Por exemplo, ao planejar um teste de campo, o testador pode pensar: que tipo de obstáculo ou de posicionamento de obstáculos devem ser incluídos no cenário de modo que um método não exercitado nas sessões de testes anteriores pode ser exercitado agora? Ou então, quando se analisa a cobertura de um teste ser capaz de relacionar os requisitos de teste que não foram cobertos (instruções, nós, arestas, etc) com as características do sistema que ainda não foram testadas.

Nesse sentido, esta tese propõe uma abordagem para apoio ao teste estrutural de softwares controladores de veículos autônomos em complementação ao teste de campo de veículos autônomos, baseado no esquema da Figura 5.1. Para isso, foi proposto um modelo para oferecer suporte ao teste estrutural de veículos autônomos que considera a geração de novos logs de teste bem como uma ferramenta que implementa o modelo proposto.

Programas para veículos autônomos passam por refinamentos sucessivos e, muitas vezes, partes do código mudam ou mesmo todo o código de uma unidade, gerando, assim, um grande número de versões do programa e logs de cenários de teste. Para gerenciar o desenvolvimento neste domínio de software é muito importante não só usar ferramentas de teste, mas várias outras ferramentas como controladores de versão, gerenciadores de configuração e analisadores estáticos. Além disso, também é importante relacionar as versões e logs de programas com cenários de testes executados com cada versão e também com a cobertura obtida. Esses recursos não são normalmente oferecidos pelas ferramentas referidas na Seção 3.3.2. Para apoiar o teste estrutural neste domínio, a autora desta tese criou um modelo formalizando o esquema da Figura 5.1. A Figura 5.2 apresenta a representação dos principais conceitos e artefatos envolvidos no modelo criado. A parte hachurada do modelo é apresentada com mais detalhes nos próximos capítulos.

Versões de programas são definidas como implementações específicas de um programa (ou especificação) que diferem em alguma parte do código, mas mantêm os mesmos elementos arquitetônicos. Assim, por exemplo, uma alteração de parâmetro não cria uma nova versão, mas a utilização de uma câmera diferente para ajudar a controlar um veículo cria uma nova versão do programa. Cenários de teste são planos de teste com o objetivo específico de testar subconjuntos de recursos (funcionalidades) do programa em teste e sujeitos a determinadas condições. Os cenários são criados por um testador com o objetivo de testar uma versão do programa.

Logs são registros de dados de entrada que podem ser capturados durante um teste de campo de uma versão de um programa controlador de veículo autônomo para um cenário particular (*LogTC*) ou versões modificadas a partir de *LogTC* (*LogGerado*). Os dados de

Figura 5.2: Um modelo para apoiar uma estratégia de teste estrutural

teste capturados em um log (*LogTC*) são as entradas recebidas pelo veículo como, por exemplo, imagens de câmeras, nuvens de pontos e posição GPS. Esses dados geralmente são capturados automaticamente usando dispositivos de instrumentação disponíveis no ambiente do sistema.

Os logs também podem ser particionados em trechos de diferentes tamanhos e agrupados de acordo com determinada heurística (ver Capítulo 6). Um trecho é, portanto, um subconjunto ordenado de *LogTC*. *LogGerado* é um subconjunto de trechos de *LogTC* cujas entradas foram modificadas por alguma estratégia de combinação e/ou mutação, seja aplicando diretamente a estratégia (ver Capítulo 7) ou utilizando algum algoritmo de geração de dados de teste. Nesta tese, um algoritmo genético foi projetado para gerar casos de teste com base em uma população inicial definida por uma heurística e com base em diferentes estratégias de combinação e mutação definidas (ver Capítulo 8).

Os logs são definidos como:

- $log_{v,s}$ é um arquivo que contém as entradas de dados para o teste da versão v de um programa para o cenário s . Nota-se que $log_{v,s}$ também pode ser um registro gerado ou por estratégias de combinação e mutação ou automaticamente por um algoritmo de geração de dados de teste.

Com base no modelo, é possível armazenar informações sobre o teste que podem ser tratadas por uma ferramenta que apoia o desenvolvedor e o testador a respeito da análise dos resultados de um teste de campo a partir dos pontos de vista estrutural e de cobertura

e usar essas informações para depurar e aperfeiçoar o programa (versão) em teste. A cobertura obtida em um teste é definida como:

- $cob_{v,s,c}$ cobertura da versão v do programa executado ou simulado para o cenário s e o critério de teste c .

Considerando a definição de um critério de cobertura de c , que pode ser a execução de todas as declarações de um programa ou todos os nós ou arestas do gráfico de controle de um programa, a ferramenta pode, por exemplo, apoiar a seguinte análise comparativa para qualquer critério de cobertura c :

- $cob_{v_1,s,c}; cob_{v_2,s,c}; \dots; cob_{v_n,s,c}$: análise da cobertura obtida com a execução de diferentes versões de um programa v sob um mesmo cenário.
- $cob_{v_1,s_1,c}; cob_{v_1,s_2,c}; \dots; cob_{v_1,s_m,c}$: análise da cobertura obtida com a execução da mesma versão de um programa v sob diferentes cenários.

A Figura 5.3 mostra um gráfico com um modelo de objeto genérico para o modelo descrito acima, para um programa P . Dado este gráfico, pares $\langle COB_{L_1}, COB_{L_2} \rangle$ e $\langle COB_{L_3}, COB_{L_4} \rangle$, por exemplo, representam uma sequência de cobertura estrutural (de diferentes critérios) de dois testes da mesma versão e mesmo cenário; o par $\langle COB_{L_1}, COB_{L_6} \rangle$ representa uma sequência de cobertura de dois testes da mesma versão para diferentes cenários; e o par $\langle COB_{L_5}, COB_{L_1} \rangle$ representa uma comparação da cobertura de dois testes de uma mesma versão, mas uma gerada por um teste de campo e o outra gerada automaticamente por um algoritmo que usou V_1 como semente inicial. Também é possível analisar a cobertura da execução de apenas de um subconjunto de trechos dos logs ou de logs gerados automaticamente. Uma outra possibilidade seria analisar a cobertura de diferentes versões sob diferentes cenários (se isso pode ser útil). A análise também pode ser feita por pares de cobertura ou para múltiplas coberturas, dependendo da implementação da ferramenta.

5.3 Implementação da ferramenta desenvolvida

O modelo apresentado na Figura 5.2 foi implementado como base subjacente de uma ferramenta para apoiar os testes de campo de veículos autônomos, tanto na análise dos resultados do teste de campo como para a geração de outros logs com os objetivos de diminuir o custo dos testes e de aumentar a cobertura. A essa fase do teste, conforme discutido na Seção 4.4, é atribuído o nome de pós-análise (ou pós-processamento). O nome da ferramenta é PATeCa (Pós-Análise de Testes de Campo).

Figura 5.3: Um modelo de objetos derivado do metamodelo apresentado na Figura 5.2.

O diagrama de casos de uso apresentado na Figura 5.4 mostra as funcionalidades que a ferramenta oferece ao testador. Nota-se na figura que existe um caso de uso básico do tipo CRUD (Larman, 2007) – Gerenciar Teste de Campo – e quatro casos de uso que representam as funcionalidades principais da ferramenta. O primeiro deles, Visualizar Cobertura, é apresentado em detalhes neste capítulo, na Seção, 5.4 e os demais são apresentados nos Capítulos 6, 7 e 8. Conforme dito anteriormente, os conceitos do modelo que não estão envolvidos nas funcionalidades descritas neste capítulo estão hachurados na Figura 5.2. Como a PATECa trata-se de um protótipo, decidiu-se implementar inicialmente apenas os critérios todos-nós e todas-arestas do teste estrutural de software.

A Figura 5.5 apresenta a arquitetura da ferramenta. Ela usa um SGBD para armazenar os dados relativos ao domínio da ferramenta e a infraestrutura do sistema operacional para armazenar os logs. O módulo de cálculo de cobertura utiliza a ferramenta GCov (Team, 2014) para gerar os registros de cobertura de cada classe do programa sob teste e que são gravadas em arquivos com a extensão **.gcno* e **.gcda*. A PATECa utiliza então uma versão adaptada da ferramenta Trucov (Terry, 2014) para analisar os arquivos gerados pelo GCov e calcular a cobertura de nós e arestas, gravando essas informações em arquivos. Em seguida, a ferramenta reúne todas essas informações de cobertura em arquivos XML a fim de serem utilizados pelo módulo de Visualização. Uma das visualizações disponíveis no módulo de visualização é a da hierarquia de classes e métodos e, para isso, a ferramenta Prefuse (Prefuse, 2014) é utilizada. A ferramenta também possui uma interface gráfica que facilita a interação do usuário com seus módulos.

O módulo de Reprodução/Simulação é responsável por reproduzir um *LogTC* ou simular um *LogGerado*. Ele interage com o módulo de Cálculo de Cobertura que, por sua vez, gera os grafos de controle e arquivos XML com informações de nós e arestas que fo-

Figura 5.4: Caso de uso da ferramenta

Figura 5.5: Arquitetura da ferramenta

Tabela 5.1: Esforço de implementação da ferramenta

Linguagem	Núm. arquivos	LOC
Java	106	22655
Python	34	2056
C++	19	2000
Total	159	26711

ram cobertos. Os resultados das simulações/reproduções são registrados pela ferramenta. Na implementação atual, simulações duplicadas para a mesma versão de um programa, cenário e log geram simulações idênticas, mas são diferenciadas por uma marcação de tempo. As simulações são escolhidas pelos cenários ou logs. O módulo de Visualização é responsável por exibir as análises de cobertura da ferramenta. As visualizações não são armazenadas e podem ser geradas *online*. As Figuras 5.6 e 5.7 apresentam, respectivamente, as telas de Gerenciamento de Cenários e de Reprodução/Simulação da PATeCa. Logs gerados automaticamente a partir de outros logs podem ser armazenados e gerenciados pela ferramenta e suas coberturas analisadas de acordo com as necessidades do testador. O único *overhead* desta abordagem é o uso do ROS para armazenar as entradas em um log.

A ferramenta executa nos sistemas operacionais Linux e a linguagem alvo do programa sob teste é C++. O ambiente ROS (ver Seção 4.2.6) é utilizado para a criação de logs de testes que podem ser importados para a ferramenta. Esses registros são obtidos no ambiente da ferramenta a partir de testes de campo que podem ser utilizados como entradas para as execuções simuladas de diferentes versões do programa e as coberturas obtidas podem ser visualizadas e analisadas. O contexto de cada sessão de teste pode ser registrado, indicando as identificações do programa e da versão, o testador e o cenário usado. A ferramenta foi implementada utilizando principalmente a linguagem Java. Devido às facilidades de integração com as ferramentas do ROS, também foram utilizadas as linguagens Python e C++, conforme apresentado na Figura 5.5. A Tabela 5.1 apresenta o número de linha de código (LOC) e o número de arquivos gerados na implementação da ferramenta.

5.4 Uso da ferramenta para visualização da cobertura e análise do teste de campo

Para ilustrar a ferramenta proposta, foram realizados testes de campo envolvendo uma das versões do programa para navegação e desvio de obstáculos descrito na Seção 4.3. O objetivo geral do teste de campo foi o de chegar a um ponto predeterminado, desviando de obstáculos. O teste de campo foi realizado considerando dois cenários: em um terreno

CAPÍTULO 5. AMBIENTE DE APOIO AO TESTE DE CAMPO DE VEÍCULOS AUTÔNOMOS

Figura 5.6: Tela de gerenciamento de cenários da PATECa

Figura 5.7: Tela de reprodução/simulação da PATECa

estreito e em um amplo. No terreno estreito era esperado que o veículo seguisse uma trajetória em linha reta e obstáculos (cones de trânsito) foram colocados no caminho durante o teste de acordo com a intuição do testador enquanto o veículo se deslocava (S1). No terreno amplo, o veículo devia fazer uma curva e dois cones foram anteriormente colocados em lugares estratégicos (S2), como mostrado na Figura 5.8. Os dois cenários de teste foram realizados dentro do campus da universidade. Um log foi gerado para cada um dos testes de campo realizados.

Figura 5.8: Caminho realizado pelo veículo autônomo durante o teste de campo

Baseado no modelo proposto, uma das responsabilidades da ferramenta é realizar o gerenciamento dos testes de campo. Dessa forma, as informações relacionadas aos testadores, programas e versões, cenários de teste e os logs gerados pela execução do teste de campo nesses cenários foram informados na ferramenta e armazenados no banco de dados, conforme pode ser visto na Figura 5.6. O próximo passo então foi reproduzir esses testes de campo a fim de calcular a cobertura alcançada por eles. Realizar o cálculo da cobertura *offline* reproduzindo o log de teste evita *overhead* de instrumentação da versão do programa durante a execução do teste de campo. A Figura 5.9 apresenta o exemplo da reprodução de um teste de campo.

Após a execução do programa, a ferramenta calcula o relatório de cobertura obtido para cada classe e também para cada método do software sob teste. Para isso, utiliza como base a ferramenta GCov (Team, 2014) do GCC. Em seguida, gera o grafo de cobertura utilizando como base a ferramenta Trucov (Terry, 2014). Tanto o relatório quanto o grafo de cobertura podem ser visualizados considerando a execução de todos os logs em conjunto ou separados por log. Foram implementados dois critérios de teste estruturais: todos-nós e todas-arestas. Uma decisão de projeto tomada sobre representar em cores a cobertura foi agrupar a intensidade de execuções de instruções do programa em cinco níveis de verde em que o maior grupo de cobertura tem a cor mais escura. O número de níveis de intensidade pode ser aumentado ou diminuído, conforme necessário e o estilo de trabalho do testador. Os nós do grafo coloridos em amarelo representam os nós que

Figura 5.9: Exemplo da reprodução de um teste de campo

tenham sido exercitados, mas tem pelo menos uma das arestas não exercitadas pelo teste. Os nós vermelhos representam nós e as arestas não exercitadas pelo teste.

A Figura 5.10 mostra uma visão abstrata da cobertura obtida após os testes de campo, utilizando os cenários S1 (canto superior esquerdo) e S2 (canto superior direito). Na parte superior da Figura 5.10 notam-se oito classes que compreendem todo o projeto e dentro delas os seus métodos. Dois métodos apresentam uma pontuação inferior da cobertura, como pode ser visto pela cor verde claro. As classes do canto superior direito da figura foram escolhidas para serem analisadas mais detalhadamente. Os dois diagramas da parte inferior da figura mostram uma visão mais detalhada das classes.

Figura 5.10: Visualização da cobertura de projeto e de classe

A Figura 5.11 mostra parcialmente a cobertura do gráfico de controle para ambos os cenários. Pode-se notar que uma parte importante do gráfico direito não foi exercitada pelo cenário S2, que é marcado pela cor vermelha. Voltando ao código, pode-se notar que esta parte do código está relacionada com a decisão de desviar de um obstáculo. Esta parte não foi exercitada porque o GPS foi desativado neste cenário, o que mudou os dados de entrada. Seria possível criar e visualizar uma versão agregada de todos os testes de campo desta versão e então essa parte do programa apareceria em verde e o testador poderia ficar satisfeito com esse resultado, pois pelo menos um teste cobriu essa parte do código.

Pode-se ver logo abaixo dessa parte do gráfico três nós de decisão (em amarelo) que não tiveram ambos os seus ramos exercitados. Pode-se concluir que estes dois cenários não foram suficientes para chegar a uma cobertura satisfatória e novos cenários precisariam ser criados ou aquele já criado poderia ser modificado. Em um grafo agregado esses nós ainda apareceriam como amarelos, indicando que estes dois testes de campo não foram suficientes para exercitar esses nós.

Figura 5.11: Visualização da cobertura de um programa para duas versões diferentes do mesmo programa e dois cenários diferentes.

Uma análise visual como a apresentada na Figura 5.11, examinando lado a lado a cobertura dos testes de acordo com seus cenários ou versões também pode ser feito no nível do código, conforme pode ser visto na Figura 5.12 em que as linhas em vermelho indicam que houve diferenças na cobertura entre um log e outro (em um log a linha foi executada e no outro não foi).

5.5 Considerações finais

Este capítulo apresentou um ambiente de apoio a pós-análise dos dados coletados a partir de testes de campo de veículos autônomos. Foi apresentado um modelo e uma ferramenta que implementa esse modelo. As funcionalidades da ferramenta apresentada nesta seção

Figura 5.12: Visualização da cobertura de um programa para dois logs diferentes.

dizem respeito aos passos 1 e 2 da Figura 5.1 (reproduzir o experimento a partir de uma coleção de logs e obter um novo relatório de cobertura, respectivamente). No entanto, essa coleção de logs que a priori foi obtida a partir dos experimentos de campo, deverá ser formada também pelos logs obtidos do passo 3 da Figura 5.1 a fim de aumentar a cobertura inicialmente alcançada. As estratégias para que isso aconteça bem como os detalhes da implementação dessa ferramenta são apresentadas nos capítulos a seguir.

Heurísticas para a redução de dados de teste de softwares controladores de veículos autônomos

6.1 Considerações iniciais

Os logs produzidos pelos testes de campo podem conter milhares de entradas. No entanto, se não há mudanças no trajeto é possível que o programa execute sempre os mesmos comandos computacionais. Sendo assim, em vez de usar o log todo, apenas um subconjunto dele pode produzir o mesmo efeito com cobertura similar. Como continuação do modelo proposto no capítulo anterior, o próximo passo é gerar outros dados de entrada com o objetivo de melhorar a cobertura de teste de acordo com os critérios estruturais. Algoritmos de geração são baseados em uma população inicial (aleatória ou não) e a qualidade dessa população pode ser crucial para o sucesso da busca (Gordon e Arcuri, 2012). Sendo assim, para a geração proposta nesta tese, é interessante que um conjunto reduzido de dados de teste representativos dos dados esteja contido em um log. Para que isso seja possível, foi feita uma análise de cinco logs obtidos a partir de cinco testes de campo para os dois métodos mais complexos do programa apresentado na Seção 4.3. A partir dessa análise, foram definidas e avaliadas seis heurísticas baseadas em critérios estruturais e uma heurística aleatória, conforme apresentadas neste capítulo. Essas heurísticas são

avaliadas com base em simulações, buscando minimizar o número de dados de entrada e maximizar a qualidade da população com base na cobertura de nós e de arestas obtida.

As propostas apresentadas neste capítulo correspondem ao caso de uso “Selecionar trechos discriminatórios” apresentado no diagrama de caso de uso da Figura 5.4 do Capítulo 5. Conforme essa figura, é possível realizar as análises de cobertura discutidas também no Capítulo 5, tanto para os trechos particionados do log quanto para o subconjunto de trechos gerados. O conjunto reduzido produzido pode ser usado independentemente, como para teste de regressão, ou então ser usado como a população inicial de um algoritmo de geração de casos de teste, como parte do caso de uso “Gerar logs usando algoritmos de busca”.

Este capítulo está organizado da seguinte forma: a Seção 6.2 apresenta as análises dos logs coletados em cinco testes de campo; na Seção 6.3 são definidas e analisadas heurísticas para a seleção de um subconjunto de logs que podem ser utilizados como população inicial em algoritmos de busca; na Seção 6.4 são descritos em detalhes os conceitos e a implementação das funcionalidades referentes a este capítulo do modelo e da ferramenta propostos no Capítulo 5; por fim, na Seção 6.5 são apresentadas as considerações finais.

6.2 Análise de cobertura de testes de campo

Inicialmente foram conduzidos cinco testes de campo com o veículo CaRINA I e o software controlador descrito na Seção 4.3, a fim de capturar dados de entrada que foram armazenados em cinco logs. Os testes de campo realizados foram:

1. Veículo andando em linha reta em local onde não há outros veículos e nem pessoas transitando. O veículo foi desligado do modo automático e andou em direção contrária ao ponto de destino.
2. Veículo andando em linha reta, sem carros transitando mas uma pessoa apareceu durante o trajeto. Foi realizado uma curva.
3. Veículo andando em linha reta com outros veículos transitando. O veículo foi desligado do modo automático e andou em direção contrária ao ponto de destino.
4. Veículo andando em linha reta com outros veículos transitando.
5. Veículo andando em um terreno com descida e com subida.

A análise foi realizada tendo como base os dois métodos principais do programa: *process()*, da classe *cObstacleAvoidance*, e *calcule()*, da classe *cVFH*, pois os demais métodos são mais simples e pequenos, atingindo facilmente altos níveis de cobertura de

CAPÍTULO 6. HEURÍSTICAS PARA A REDUÇÃO DE DADOS DE TESTE DE SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

Tabela 6.1: Dados dos logs e das coberturas dos cinco testes de campo

Log	Tamanho (NP)	Trechos - 5NP	Trechos - 10NP	Process-Nós	Process-Arst	Calcule-Nós	Calcule-Arst
1	1412	283	142	60%	46%	91%	74%
2	680	137	69	71%	56%	93%	78%
3	1951	391	196	70%	54%	93%	77%
4	2100	421	211	71%	56%	90%	74%
5	2487	498	249	71%	56%	95%	81%
media	1726	346	173	68,6%	53,6%	92,4%	76,8%

nós e arestas. As coberturas conseguidas nos cinco testes de campo para os métodos *process()* e *calcule()* são mostradas na Tabela 6.1. O tamanho dos logs registrados na tabela indica o número de nuvens de pontos que foram usadas pelo programa durante o teste. Isso corresponde principalmente ao tempo gasto para fazer os percursos do teste. Os logs podem ser usados para executar novamente o programa de forma simulada, sempre que necessário, e foram usados neste trabalho para realizar estudos sobre as coberturas obtidas com vistas a selecionar um subconjunto das nuvens de pontos otimizado em relação ao tamanho e à cobertura obtida com eles.

Após obtidos os logs, a análise foi iniciada. Primeiramente, eles foram particionados em trechos (ou subsequências) contendo aproximadamente cinco nuvens. Em seguida, outro particionamento foi realizado e, dessa vez, contendo aproximadamente dez nuvens. Os números de trechos encontram-se, respectivamente, na terceira e quarta colunas da Tabela 6.1. O programa foi executado de forma simulada usando os trechos como entrada e as coberturas de nós e de arestas de cada trecho foram calculadas. A razão para isso vem da intuição física de como o programa se comporta quando executado: de um modo geral, em cada ciclo de leitura da nuvem de pontos, se não há obstáculos no caminho que precisem de desvios e mudanças de velocidade, é natural que o programa execute os mesmos comandos computacionais e condições. Foram realizadas duas análises: uma com trechos contendo cinco nuvens de pontos e outra com trechos contendo dez nuvens de pontos. Os números de nuvens de pontos foram escolhidos arbitrariamente para que representassem uma pequena parte do percurso realizado durante o teste. Como limite mínimo, cada trecho poderia ter apenas uma nuvem de pontos. As subseções seguintes apresentam os resultados dessas análises.

6.2.1 Análise para os logs particionados em trechos contendo cinco nuvens de pontos

Os gráficos apresentados nas Figuras 6.1 e 6.2 mostram as coberturas individuais de cada trecho particionado em tamanho 5 do $\log_2$ para os métodos *process()* e de cada trecho do $\log_5$ para o método *calcule()*, respectivamente. Gráficos semelhantes foram feitos para os outros três logs mas por serem muito parecidos optou-se por apresentar apenas um deles escolhido arbitrariamente. A ordem em que os trechos são apresentados é a mesma em

CAPÍTULO 6. HEURÍSTICAS PARA A REDUÇÃO DE DADOS DE TESTE DE SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

que aparecem no log completo, isto é, na ordem do percurso do teste de campo. Pode-se notar que:

- há uma correspondência entre as coberturas de arestas e nós e na forma geral do contorno das barras;
- a cobertura de nós dos métodos *process()* e *calcule()* situa-se em torno de 50%, comparadas com cerca de 70% e 90% do log completo (Tabela 6.1);
- a cobertura de arestas dos métodos *process()* e *calcule()* situa-se em torno de 40% a 50%, comparadas com cerca de 50% e 70% do log completo (Tabela 6.1).

Figura 6.1: Cobertura de nós (a) e arestas (b) do método *process()* com o log2 particionado em tamanho 5

Figura 6.2: Cobertura de nós (a) e arestas (b) do método *calcule()* com o log5 particionado em tamanho 5

No segundo passo foi feita uma análise mais detalhada das coberturas de cada trecho, verificando-se quais nós ou arestas ele cobria e quais nós ou arestas adicionais eram cobertos que já não haviam sido cobertos por algum dos trechos anteriores. Os gráficos foram então reorganizados, mostrando a contribuição de cada trecho para a cobertura acumulada (ou agregada) de todos os trechos.

Na Figura 6.3 é mostrado um gráfico com a cobertura acumulada para o critério todos-nós dos trechos do log2 para o método *process()*. Na Figura 6.4 mostra-se a cobertura acumulada para o método *calcule()* usando o log5. Nota-se que em ambas as tabelas

Figura 6.3: Cobertura acumulada de nós do método *process()* para os trechos particionados em tamanho 5 do log2

Figura 6.4: Cobertura acumulada de arestas do método *calcule()* para os trechos particionados em tamanho 5 do log5

há alguns trechos que têm uma contribuição maior para a cobertura, gerando um salto na altura do diagrama (coloridos em verde). Esses trechos são interpretados como aqueles em que houve alguma mudança nas condições do percurso que levaram o programa a percorrer outros caminhos em seu fluxo de execução. Esses trechos receberam o nome de trechos discriminatórios. Conforme a intuição inicial, eles ocorrem em pequeno número e podem ser bons candidatos para gerar a população inicial buscada. Com base neles foram criadas e testadas algumas heurísticas. Deve-se notar também que ao executar os trechos, para cada um deles o programa é reexecutado e, portanto, parte do estado inicial. Isso pode produzir resultados e coberturas um pouco diferentes da execução do log completo.

6.2.2 Análise para os logs particionados em trechos contendo dez nuvens de pontos

De forma análoga, os gráficos apresentados nas Figuras 6.5 e 6.6 mostram as coberturas individuais de cada trecho particionado em tamanho 10 do log3 para os métodos *process()* e de cada trecho do log1 para o método *calcule()*, respectivamente.

Figura 6.5: Cobertura de nós (a) e arestas (b) do método *process()* com o log3 particionado em tamanho 10

Figura 6.6: Cobertura de nós (a) e arestas (b) do método *calcule()* com o log1 particionado em tamanho 10

A cobertura acumulada para o critério todos-nós dos trechos do log3 para o método *process()* é mostrada na Figura 6.7. Na Figura 6.8 mostra-se a cobertura acumulada para o método *calcule()* usando o log1.

Figura 6.7: Cobertura acumulada de nós do método *process()* para os trechos particionados em tamanho 10 do log3

Figura 6.8: Cobertura acumulada de arestas do método *calcule()* para os trechos particionados em tamanho 5 do log1

6.3 Heurísticas para criação da população inicial: propostas e análises

Com base nas análises apresentadas na seção anterior, foram definidas sete heurísticas a serem investigadas para verificar a cobertura que elas proporcionam, considerando também o número de trechos como um fator a ser minimizado. As heurísticas são as seguintes:

- H1 – trechos discriminatórios de todos-nós.
- H2 – trechos discriminatórios de todas-arestas.

- H3 – trechos discriminatórios de todos-nós mais trechos inicial e final.
- H4 – trechos discriminatórios de todas-arestas mais trecho inicial e final.
- H5 – trechos discriminatórios de todos-nós de todos os métodos sob teste.
- H6 – trechos discriminatórios de todas-arestas de todos os métodos sob teste.
- H7 – trechos escolhidos aleatoriamente.

Uma heurística nula (H0), ou fator de controle, é a cobertura obtida com o log original completo, que deve funcionar como um limitante superior. Os valores são os da Tabela 6.1. A heurística aleatória (H7) funciona como controle e deveria produzir resultados inferiores às demais. Ela pode ser implementada, por exemplo, pedindo ao testador para que escolha alguns trechos aleatoriamente.

As heurísticas H1 e H2 são focadas em maximizar a cobertura de um método para um critério de teste. As heurísticas H3 e H4 são uma composição das duas primeiras com os trechos inicial e final, considerando que nestes pontos podem ocorrer computações diferentes do meio do percurso. A quinta heurística (H5) junta os trechos discriminatórios do critério todos-nós para todos os métodos sob teste, com o objetivo de maximizar a cobertura de todos os métodos envolvidos no teste. A heurística H6 é semelhante à H5, mas para o critério todas-arestas.

O Algoritmo 2 foi proposto para reduzir o tamanho dos logs de entrada obtidos a partir dos testes de campo. Na primeira parte do algoritmo, o log obtido do teste de campo é dividido em trechos contendo entradas coletadas em cada ciclo de execução do programa sob teste. No caso do exemplo utilizado nesta tese, os elementos atômicos são nuvens de pontos. Conforme dito anteriormente, o tamanho de cada trecho é variável e pode ser definido empiricamente. Isso é feito na linha 4 do algoritmo.

O programa sob teste é executado com cada segmento como entrada e a cobertura obtida para cada método de interesse e critério é calculada e cada requisito de teste satisfeito é armazenado. Em seguida, para cada trecho, na mesma ordem que aparecem no log original, é calculado quantos requisitos foram satisfeitos e que não haviam sido satisfeitos nos trechos anteriores. Isso é feito nas linhas 5-12 do algoritmo. Se o trecho atual satisfaz pelo menos um requisito novo e que não havia sido satisfeito pelos trechos anteriores, ele é armazenado em uma variável local que armazena os trechos discriminatórios. A segunda parte do algoritmo acrescenta os trechos inicial e final ao conjunto de trechos discriminatórios, caso a heurística seja H3 ou H4.

Nos estudos apresentados a seguir foram definidos os conjuntos de trechos produzidos pelas heurísticas e depois eles foram usados como entrada para o programa controlador do veículo autônomo do ICMC e as coberturas obtidas foram analisadas, considerando

Algorithm 2 Algoritmo para selecionar trechos discriminatórios para as heurísticas H1-H6

Entrada:

logTC: log gerado a partir do teste de campo
metodosSobTeste: lista de métodos considerados. Se *heuristica* = H1,..., H4, contém somente um método.
requisito: nós ou arestas
numNP: número de nuvens de pontos *entradas* em cada trecho
versao: versão do programa sob teste
heuristica: H1, H2,..., H6

Variáveis:

reqCobertos: conjunto dos requisitos de teste cobertos pelos trechos
reqCoberto: um elemento de *reqCobertos*
trechosDiscriminatorio: conjunto de trechos discriminatórios calculados
cjtoTrechos: lista de segmentos particionados a partir de *logTC*
trecho: um elemento de *cjtoTrechos*

Funções e procedimentos:

particionarLog(): procedimento responsável por particionar *logTC* em trechos contendo *numNP* nuvens de pontos. O conjunto trechos é retornado de maneira ordenada
simular(): procedimento que executa a *versao* do programa sob teste tendo como entrada um determinado trecho

```

/**** Inicialização e configuração ****/
1: reqCobertos ← NULO
2: trechosDiscriminatorio ← NULO
3: Calcular o GFC dos métodos sob teste
/**** Seleção ****/
4: cjtoTrechos ← particionarLog(numNP)
5: Para trecho em cjtoTrechos faça
6:   simular(trecho)
7:   reqCobertosTrecho ← Computar os requisitos cobertos de trecho, de acordo com requisito
8:   Se  $\exists$  reqCoberto  $\in$  reqCobertosTrecho | reqCoberto  $\notin$  reqCobertos então
9:     reqCobertos.add(reqCoberto);
10:    trechosDiscriminatorios.add(trecho);
11:   Fim Se
12: Fim Para
13: Se (heuristica = H3) OU (heuristica = H4) então
14:   trechosDiscriminatorios.add(trechos.first());
15:   trechosDiscriminatorios.add(trechos.last());
16: Fim Se
17: Retorna trechosDiscriminatorios como LogGerado
```

os logs, os métodos *process()* e *calcule()* e os critérios todos-nós e todas-arestas. Como exemplo, considerando trechos particionados em tamanho cinco, o log 2 e a heurística H1 para *process()* gera o conjunto de trechos {1,2,66,67,78}, conforme a Figura 6.3, para a heurística H4 seria acrescentado o trecho 137, pois o primeiro já está incluído. Para a heurística H5, e o log 2 o conjunto seria {1,2,9,11,66,67,68,75,83,85}. H6 teria o mesmo conjunto que o H5 para esse log.

6.3.1 Trechos contendo cinco nuvens de pontos

Foram gerados setenta conjuntos de trechos (um para cada método (2), para cada heurística (7), para cada log (5)). O programa foi executado em modo de simulação e as coberturas obtidas para os dois métodos considerados foram calculadas. As médias das coberturas é apresentada na Figura 6.9 (a). A heurística H0 é o log completo. É fácil perceber que H7 (aleatório) sempre produziu resultados piores, e que H0 é sempre superior, como esperado. Entretanto, nota-se que os resultados conseguidos com os conjuntos de trechos discriminatórios estão muito próximos do resultado com o log completo, apesar do tamanho muito menor de trechos/nuvens de pontos.

Figura 6.9: Cobertura média obtida com as sete heurísticas propostas (a) e média do tamanho dos conjuntos de trechos (b) para trechos particionados com tamanho 5

A Figura 6.10 apresenta a cobertura obtida pelo critério todos-nós do método *process* para todos os logs. Nota-se que a cobertura obtida pelos logs 1 e 3 são menores que dos demais logs. Isso se deve ao fato de que nesses logs o veículo foi desligado do modo automático e fez um trajeto contrário ao ponto de chegada definido e, dessa forma, o programa nunca entrou no estado *On goal* (Ver Seção 4.3). A Figura 6.11 apresenta a cobertura obtida pelo critério todas-arestas do método *calcule* para todos os logs.

Figura 6.10: Cobertura para o critério todos-nós do método *process* e logs particionados em trechos de tamanho 5

Figura 6.11: Cobertura para o critério todas-arestas do método *calcule* e logs particionados em trechos de tamanho 5

Considerando-se também os resultados obtidos por log, nota-se que pode haver algumas diferenças significativas da cobertura obtida, mas a média dos resultados dá uma visão consistente dos resultados obtidos. A análise dos dados obtidos permitiu concluir que os trechos iniciais são muito importantes, pois os trechos 1 ou 2 (ou ambos) apareceram como discriminatórios em todas as heurísticas. O trecho final não pareceu importante, haja vista os resultados das heurísticas H3 e H4.

As heurísticas H1 e H3 geralmente têm o menor número de trechos em seu conjunto, conforme se nota na Figura 6.9 (b), mas não produzem os melhores resultados. Nota-se que o conjunto de trechos de H5 (todos-nós) é um subconjunto de H6. De um modo geral, a heurística H6, que reúne os trechos discriminatórios para o critério todas-arestas dos dois métodos sob teste, produziu os melhores resultados, mas é a que contém o maior número médio de trechos. Esse número, entretanto, é relativamente baixo.

Concluindo, nota-se que as heurísticas que envolvem a seleção dos trechos discriminatórios de todas-arestas (H2 e H6) obtiveram os melhores resultados. Enquanto a heurística H6 tem uma cobertura levemente superior a H2, esta última possui um conjunto menor de trechos. Assim, caberia ao testador decidir qual heurística utilizar levando em consideração o que seria mais importante: maior cobertura ou número menor de trechos.

6.3.2 Trechos contendo dez nuvens de pontos

O mesmo algoritmo foi aplicado ao mesmo programa e o mesmo conjunto de logs mas para cada trecho contendo 10 nuvens de pontos. A Figura 6.12(a) apresenta a média das coberturas obtidas para cada log e a Figura 6.12 (b) a média dos conjuntos dos trechos em cada log gerado produzido. Pode-se notar que houve diferenças significativas para as heurísticas dos logs 1 e 3 para o método *process()* e em alguns casos da heurística H7, como era esperado. As médias dos valores das coberturas, no entanto, como mostrado na Figura 6.12(a), são muito similares às médias de cobertura obtidas por trechos contendo cinco nuvens de pontos, como mostrado na Figura 6.9(a). Os tamanhos do conjunto reduzido são, em média, aproximadamente 40% menor que o tamanho do conjunto para trechos contendo 5 nuvens de pontos, mas deve-se notar que cada trecho do último conjunto contém duas vezes o número de nuvens de pontos. Essas observações indicam que parece não haver nenhuma vantagem no aumento do tamanho de trechos.

A Figura 6.13 apresenta as coberturas obtidas pelo critério todos-nós do método *process* para todos os logs. Nota-se que, assim como para os trechos particionados em cinco nuvens de pontos, os logs 1 e 3 obtiveram os piores resultados. A Figura 6.14 apresenta a cobertura obtida pelo critério todas-arestas do método *calcule()* para todos os logs. Nesse caso, os logs obtiveram coberturas parecidas.

Figura 6.12: Cobertura média obtida com as sete heurísticas propostas (a) e média do tamanho dos conjuntos de trechos(b) para trechos particionados com tamanho 10

Figura 6.13: Cobertura para o critério todos-nós do método *process* e logs particionados em trechos de tamanho 10

Figura 6.14: Cobertura para o critério todas-arestas do método *calcule* e logs particionados em trechos de tamanho 10

As Figuras 6.15 e 6.16 apresentam uma comparação entre os resultados obtidos com trechos contendo cinco e dez nuvens de pontos para o critério todas-arestas dos métodos *process* e *calcule*, respectivamente. Nota-se que para o primeiro método, o resultado é similar, mas geralmente os trechos contendo dez nuvens de pontos são 10% melhores. Para o último, os resultados são praticamente os mesmos. Essa diferença pode ser explicada pelo fato do método *process*() utilizar mais condições que envolvem as entradas, ou seja, as nuvens de pontos; além disso, o método *calcule*() já havia conseguido uma cobertura alta com trechos contendo cinco nuvens de pontos.

Figura 6.15: Média das coberturas para o critério todas-arestas do método *process* e trechos de tamanhos 5 e 10

As diferenças entre os tamanhos dos conjuntos reduzidos para os trechos contendo cinco e dez nuvens de pontos são apresentadas na Figura 6.17. Pode-se notar que o

CAPÍTULO 6. HEURÍSTICAS PARA A REDUÇÃO DE DADOS DE TESTE DE SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

Figura 6.16: Média das coberturas para o critério todas-arestas do método *calcule* e trechos de tamanhos 5 e 10

número de trechos contendo dez nuvens de pontos é cerca de 20 a 30% menor, mas o número de nuvens de pontos é cerca de 30 a 40% maior. Considerando os resultados em relação às coberturas obtidas, pode-se concluir que um número entre cinco e dez nuvens de pontos por trecho é adequado e que essa escolha também depende da sensibilidade dos métodos a serem testados em relação ao uso das variáveis de entrada.

Figura 6.17: Média do número de trechos por heurística (a) e média do número de nuvens de pontos por heurística (b)

Os resultados apresentados nesta seção mostram que com um número muito pequeno de trechos (ou nuvens de pontos), comparado ao tamanho do log original, é possível construir um conjunto reduzido de dados de teste que produzem uma cobertura para os critérios todos-nós e todas-arestas muito próxima da cobertura alcançada pelo log obtido pelo teste de campo. Esses resultados são importantes para o uso em teste de regressão e também como população inicial de algoritmos baseados em busca para a geração de dados de teste. Além disso, é possível conjecturar que resultados similares podem ser

obtidos para outros domínios de aplicação com características similares aos dos veículos autônomos, isto é, software que controla um processo físico utilizando um ciclo repetitivo no qual os dados do ambiente são lidos a fim de tomarem decisões sobre como controlar o processo físico. Por exemplo, dispositivos médicos como marca-passos e sistemas de vigilância.

6.4 Modelo e ferramenta PATeCa

A Figura 6.18 apresenta o modelo com as classes e associações utilizadas neste capítulo (parte não hachurada). O modelo considera que um log pode ser de dois tipos: log obtido a partir de um teste de campo (*LogTC*) e o log gerado de acordo com alguma heurística (*LogGerado*). Além disso, um *Log* (abstração de *LogTC* e *LogGerado*), pode ser composto de zero a n trechos, sendo n Inteiro. Os logs gerados podem ser obtidos pela composição dos trechos obtidos por alguma heurística ou pela escolha do testador.

Figura 6.18: Modelo atualizado da ferramenta PATeCa

Para oferecer suporte ao modelo ampliado proposto na Seção 6.2, a ferramenta PATeCa também foi estendida, implementando o algoritmo apresentado na Seção 6.3. A extensão da ferramenta possibilita: a) particionar logs de tamanho parametrizável; b) selecionar automaticamente os trechos discriminatórios de um conjunto de trechos de tamanho n de um log segundo uma determinada heurística; c) gerar um novo log a partir desses trechos discriminatórios ou a partir de um conjunto de trechos selecionados arbitrariamente pelo testador; d) realizar a simulação dos logs gerados; e e) calcular a cobertura dos logs gerados e compará-las, calculando também a cobertura agregada.

CAPÍTULO 6. HEURÍSTICAS PARA A REDUÇÃO DE DADOS DE TESTE DE SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

A Figura 6.19 apresenta a tela para particionamento de logs. Inicialmente o usuário seleciona qual log deseja particionar, informa o tamanho desejado de nuvens de pontos que cada trecho conterá e o diretório onde esses trechos serão armazenados. A ferramenta então particiona o log em trechos e os armazena no diretório informado. Em seguida, conforme pode ser visto na Figura 6.20, o usuário deve escolher a heurística e os métodos sob teste e a PATECa calcula automaticamente o conjunto de trechos discriminatórios, gerando então, um novo log.

Figura 6.19: Tela para particionamento em trechos

Figura 6.20: Tela para geração de logs a partir de heurísticas

6.5 Considerações finais

Neste capítulo foram propostas e avaliadas sete heurísticas para seleção de um conjunto reduzido de dados de entrada para VA, com base em logs produzidos por testes de campo. Esses conjuntos podem ser usados como população inicial ou semente para algoritmos de teste baseado em busca para posterior geração de novos logs, bem como para teste de regressão.

Foram realizadas análises da cobertura de um programa controlador de um veículo autônomo. Foram utilizados cinco logs obtidos a partir de testes de campo e, posteriormente, esses logs foram particionados e a cobertura atingida pelos subconjuntos de trechos

foi calculada. Notou-se que com um pequeno subconjunto desses trechos é possível obter uma cobertura próxima à da execução do log completo. Por exemplo, para o log5, que continha 2487 nuvens de pontos e obteve 56% de cobertura para o critério todas-arestas do método *process()*, foi possível obter uma cobertura de 48% utilizando apenas 65 nuvens de pontos.

Outro ponto interessante observado é que em um determinado log a cobertura de seu conjunto de trechos foi maior que a cobertura completa do log. Foi realizado um estudo no código fonte para determinar quais nós e arestas foram cobertos por esse trecho e que não havia sido coberto no log completo. Observou-se que era um tratamento para o caso de uma mensagem não ter sido recebida. No caso do log completo, essa mensagem era recebida desde seu início. Pode-se concluir então que a junção desses trechos trata-se de um novo log que pode ter um comportamento diferente do log a partir do qual foi gerado.

As heurísticas que envolvem os critérios todas-arestas, H2 e H6, obteve os melhores resultados. A heurística H6, em geral, tem a cobertura levemente superior que a heurística H2 mas, dependendo do número de métodos sob teste considerado, o tamanho de trechos pode ser muito superior ao da heurística H2. A escolha entre uma e outra depende de cada caso, uma vez que tenha que se considerar o que é mais importante, se uma maior cobertura do programa ou um menor número de trechos gerados.

Estratégias de combinação e mutação para geração de dados de teste para softwares controladores de veículos autônomos

7.1 Considerações iniciais

No capítulo anterior foram definidas heurísticas capazes de selecionar um pequeno subconjunto de segmentos formados por nuvens de pontos, constituindo um log que, ao ser executado, tem cobertura semelhante ao log completo. Notou-se também que a execução de apenas esse subconjunto pode ter um comportamento diferente do log original, podendo cobrir trechos ainda não cobertos por ele. Como continuação, seria interessante que, a partir desses logs gerados fosse possível obter novos logs aplicando modificações nas nuvens de pontos pertencentes a um trecho desses logs.

Este capítulo apresenta e analisa um conjunto de estratégias de combinação e mutação que podem ser usadas para gerar novos logs a partir de logs criados pelo algoritmo e pelas heurísticas apresentadas no capítulo anterior. Esta proposta corresponde ao caso de uso “Gerar logs a partir de estratégias de combinação e mutação” apresentado no diagrama de caso de uso da Figura 5.4 do Capítulo 5. Essas estratégias podem ser aplicadas indi-

vidualmente ou como parte de um algoritmo baseado em busca para geração de dados de teste, como é definido no caso de uso “Gerar logs usando algoritmos de busca”.

O capítulo está organizado da seguinte forma: a Seção 7.2 mostra as definições das estratégias de combinação e mutação; a Seção 7.3 apresenta e discute as funcionalidades de caso de uso e sua implementação na ferramenta PATeCa, bem como os conceitos e associações do modelo proposto no Capítulo 5 e que são apresentados neste capítulo; a Seção 7.4 apresenta um estudo de caso realizado com os logs obtidos a partir de cinco testes de campo e, por fim, a Seção 7.5 apresenta as considerações finais deste capítulo.

7.2 Geração de novos logs a partir de estratégias de mutação e combinação

Com a finalidade de descobrir boas estratégias para combinar nuvens de pontos, formando outras nuvens com maior probabilidade de aumentar a cobertura obtida com o log inicial, foi definido um algoritmo que usa o modelo apresentado no Capítulo 5 e gera novos logs. O algoritmo ocorre em quatro passos:

- escolhe-se um log inicial como base. Preferencialmente um log pequeno, que o testador pode criar escolhendo trechos ou gerar a partir de uma heurística;
- para cada trecho do log escolhido aplica-se uma estratégia de mutação e combinação sobre suas nuvens de pontos, gerando novas nuvens de pontos;
- uma determinada nuvem de pontos gerada é inserida (ou intercalada) no trecho, em local definido por uma estratégia, explicada a seguir, gerando assim um novo trecho;
- o log gerado é composto por esses novos trechos que contém nuvens de pontos novas ou modificadas. A cobertura obtida segundo o critério desejado é calculada executando os trechos desse log, na mesma ordem do log original.

As estratégias de mutação e combinação podem ser baseadas em intuições físicas sobre o ambiente em que o teste de campo foi conduzido, no código do programa ou então podem ser puramente teóricas. Genericamente, seja N o conjunto dos números naturais, L o log gerado por um teste de campo, L' o log base composto por um conjunto de trechos T selecionados por uma heurística H e m o número de trechos de L' . T é composto por k nuvens de pontos np , $k \in N$. Seja NP um conjunto de nuvens de pontos. Cada nuvem de pontos np é representada por uma matriz com cardinalidade $[tl, tc]$ em que tl representa o número de linhas e tc representa o número de colunas da matriz. Cada célula representa um ponto que contém os valores das coordenadas (x, y, z) com informações

CAPÍTULO 7. ESTRATÉGIAS DE COMBINAÇÃO E MUTAÇÃO PARA GERAÇÃO DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

sobre as dimensões espaciais das imagens detectadas pela câmera e também os valores das cores (r, g, b) dessas imagens.

As estratégias de mutação e combinação (E) são definidas por funções de ordem $n, n > 0$. No contexto desta tese estão sendo usados os seguintes tipos de funções:

- $E(NP) \mapsto NP$ (I)
- $E(NPXNP) \mapsto NP$ (II)
- $E(NPXNP) \mapsto (NPXNP)$ (III)

O contradomínio dessas funções contém as nuvens de ponto np' geradas pelas funções, ou transformações. De um modo geral, as mutações são do tipo I e as combinações são do tipo II e III.

Um novo log L'' é gerado a partir de L' e é composto por n trechos T' , onde T' é gerado considerando uma estratégia de mutação ou combinação E para criar novas nuvens de pontos modificadas, np' . T' é constituído de NP' . Cabe ao testador decidir como NP' será formado. Ele pode ser constituído apenas de np' , conforme apresentado na Figura 7.1 ou pode ser constituído de np intercaladas por np' , conforme apresentado na Figura 7.2. Nessas figuras, as nuvens sem preenchimento representam as nps e as nuvens com preenchimento representam as np' s. Dessa forma, o número de nuvens de pontos depende de como NP foi constituído e da estratégia adotada, variando de k até $2k$.

Figura 7.1: Esquema de geração de novos logs que considera apenas as nuvens de pontos geradas

Figura 7.2: Esquema de geração de novos logs que considera as nuvens de pontos geradas e as nuvens de pontos originais

Após a geração dos novos trechos T' , as coberturas obtidas pelo log L'' são calculadas para cada método do programa, considerando um critério de teste específico. Essa cobertura pode ser comparada com a cobertura do log base, mas mais importante, é possível verificar se algum requisito de teste novo foi coberto, isto é, que não havia sido coberto anteriormente pelo log base L' . Essa cobertura é chamada de cobertura agregada de L' e L'' . Portanto, sendo nC o número de nós cobertos, nT o número total de nós, aC o número de arestas cobertas e aT o número total de arestas em um Grafo de Fluxo de Controle, a cobertura de L' é calculada da seguinte forma:

$$Cob_{Nos}(T_1, \dots, T_n) = \left(\frac{nC}{nT}\right)$$

$$Cob_{Arestas}(T_1, \dots, T_n) = \left(\frac{aC}{aT}\right)$$

Da mesma forma, a cobertura de L'' é:

$$Cob_{Nos}(T'_1, \dots, T'_n) = \left(\frac{nC}{nT}\right)$$

$$Cob_{Arestas}(T'_1, \dots, T'_n) = \left(\frac{aC}{aT}\right)$$

Seja NC' o conjunto dos requisitos cobertos pela execução do log L' para um critério de teste, e seja NC'' o conjunto dos requisitos cobertos pela execução do log L'' para um critério de teste. A cobertura agregada de L' e L'' é:

$$Cob_{AgNos} = \frac{|NC - NC''|}{nT}$$

$$Cob_{AgArestas} = \frac{|NC - NC''|}{aT}$$

Nesta tese foram consideradas as estratégias de combinação, Seções 7.2.6 a 7.2.8, e de mutação definidas respectivamente nas Seções 7.2.1 a 7.2.5. Algumas dessas estratégias foram criadas pensando na geração de novos cenários que não haviam sido testados durante o teste de campo. A estratégia de inserção de obstáculos, por exemplo, pode criar objetos de diversos tamanhos e posicionados em lugares que não haviam sido considerados no teste de campo. A estratégia de mudança de coordenadas pode aproximar ou afastar um obstáculo mais rapidamente se a coordenada x for alterada ou, ainda, retirar e inserir um obstáculo se a coordenada z , que é utilizada para calcular a presença de obstáculos, for alterada. A estratégia de rotação vertical pode mudar a orientação do cenário original, por exemplo, se no cenário real o veículo só tenha realizado curvas à direita, aplicando essa mutação, o veículo passaria a realizar curvas à esquerda. Note que as estratégias são sempre aplicadas a trechos, mas eventualmente poderiam ser aplicadas a um log.

7.2.1 Estratégia de mutação 1: Mudança de Coordenadas - (MC)

A estratégia de mutação de mudança de coordenadas é do tipo I e é definida da seguinte forma:

$$MC(np_i(x, y, z)) \mapsto np'_i(x + cx, y + cy, z + cz)$$

para $cx, cy, cz \in N$, $i \in N, i = 1, \dots, k$ e $np_i \in T$.

Portanto, são acrescentadas aos eixos x, y, z as constantes $cx, cy, cz \in N$ a um ponto (ou pixel) da nuvem de pontos np_i . Esta estratégia desloca o ponto em suas três dimensões. As constantes podem variar em cada experimento realizado com esta função e elas podem ser fixas para todas as nuvens de pontos ou podem variar aleatoriamente. Note que se, por exemplo, cx e cy forem escolhidas como zero, então será alterada apenas a altura do objeto ao qual o ponto se refere. Se o conjunto NP' resultante da aplicação dessa estratégia for também constituído de np , np'_i é inserida logo após np_i , e o tamanho de T' é $2k$. Caso contrário, o tamanho de T' é k .

7.2.2 Estratégia de mutação 2: Rotação Vertical (RV)

A estratégia de mutação de rotação vertical é do tipo I e é definida da seguinte forma:

$RV(np_{l(i,j)}(x, y, z)) \mapsto np'_{l(i,j)}(x, y, z)$
para $np'_{l(i,j)}(x, y, z) = np_{l(i,tc+1-j)}(x, y, z)$; $i, j, l \in N$; $i = 1, \dots, tl$; $j = 1, \dots, tc$; $l = 1, \dots, k$ e $np_l \in T$

Esta estratégia realiza uma rotação de 180 graus das células da matriz considerando um eixo vertical em sua coluna central. Esta mutação é ilustrada pela Figura 7.3. Quando o número de colunas da matriz de nuvens de ponto é ímpar, a coluna central não se movimenta. Se o conjunto NP' resultante da aplicação dessa estratégia for também constituído de np , np'_i é inserida logo após np_l , e o tamanho de T' é $2k$. Caso contrário, o tamanho de T' é k .

Figura 7.3: Exemplo de rotação vertical

7.2.3 Estratégia de mutação 3: Rotação Horizontal (RH)

A estratégia de mutação de rotação horizontal é do tipo I e é definida da seguinte forma:

$RH(np_{l(i,j)}(x, y, z)) \mapsto np'_{l(i,j)}(x, y, z)$
para $np'_{l(i,j)}(x, y, z) = np_{l(tl+1-i,j)}(x, y, z)$; $i, j, l \in N$; $i = 1, \dots, tl$; $j = 1, \dots, tc$; $l = 1, \dots, k$ e $np_l \in T$

Esta estratégia realiza uma rotação de 180 graus das células da matriz considerando um eixo horizontal em sua linha central. Esta mutação é ilustrada pela Figura 7.4. Quando o número de linhas da matriz de nuvens de ponto é ímpar, a linha central não se movimenta. Se o conjunto NP' resultante da aplicação dessa estratégia for também constituído de np , np'_l é inserida logo após np_l , e o tamanho de T' é $2k$. Caso contrário, o tamanho de T' é k .

Figura 7.4: Exemplo de rotação horizontal

7.2.4 Estratégia de mutação 4: Rotação Vertical e Horizontal (RVH)

A estratégia de mutação de rotação vertical e horizontal é do tipo I é definida da seguinte forma:

$$RVH(np_{l(i,j)}(x, y, z)) \mapsto np'_{l(i,j)}(x, y, z)$$

para $np'_{l(i,j)}(x, y, z) = np_{l(tl+1-i, tc+1-j)}(x, y, z)$; $i, j, l \in N$; $i = 1, \dots, tl$; $j = 1, \dots, tc$; $l = 1, \dots, k$ e $np_l \in T$

Esta estratégia realiza uma rotação de 180 graus das células da matriz considerando tanto o eixo vertical quanto o eixo horizontal em sua coluna e linha central, respectivamente. Esta mutação é ilustrada pela Figura 7.5. Quando o número de linhas da matriz de nuvens de ponto é ímpar, a linha central não se movimenta. Analogamente, quando o número de colunas da matriz de nuvens de ponto é ímpar, a coluna central não se movimenta. Se o conjunto NP' resultante da aplicação dessa estratégia for também constituído de np , np'_l é inserida logo após np_l , e o tamanho de T' é $2k$. Caso contrário, o tamanho de T' é k .

Figura 7.5: Exemplo de rotação vertical e horizontal

7.2.5 Estratégia de mutação 5: Inserção de Obstáculos (OBS)

Esta estratégia de mutação é do tipo I e são acrescentados um ou mais objetos simulados na nuvem de pontos gerada np' . Os objetos são retangulares e podem simular postes, árvores, carros, etc. Esta estratégia deve levar em conta quatro decisões a serem tomadas, as quais são discutidas a seguir.

A primeira decisão é quanto à quantidade de obstáculos a ser inserida. A implementação pode criar sempre um número fixo de obstáculos ou criar uma quantidade aleatória limitada por um parâmetro. Esta última foi a opção usada nesta tese.

A segunda decisão é quanto à posição dos obstáculos. As posições podem ser definidas pelo testador ou serem geradas aleatoriamente. A última foi escolhida e considera uma zona delimitadora para a geração aleatória da posição inicial do objeto. Essa zona delimitadora sempre vai existir e a implementação escolhe aleatoriamente uma constante k que pode variar de 1% a 20% da menor medida da matriz e, então, desconsidera $k\%$ das primeiras linhas e $k\%$ das últimas colunas. Considera-se o ponto do canto superior esquerdo para definir se o objeto está ou não dentro da zona permitida.

Sabe-se que o programa verifica a altura dos pontos (coordenada z) para determinar a altura do obstáculo. Por isso, essa coordenada é aumentada para todos os pontos dentro dos limites do objeto inserido. Pode-se usar para isso uma constante que pode ser passada como parâmetro ou usar um valor aleatório. Optou-se por usar um parâmetro na implementação.

O tamanho máximo dos objetos também deve ser definido. A implementação pode considerar um parâmetro delimitador ou então usar alguma outra medida. Na implementação decidiu-se por usar um parâmetro que considera o percentual máximo do tamanho do objeto. Uma constante então será escolhida aleatoriamente e irá variar de 1% ao percentual informado como parâmetro. O tamanho então é calculado considerando o percentual dessas constantes aplicados às medidas de tl e tc . Se um objeto ultrapassar as bordas da matriz, esses valores serão desconsiderados e, como consequência, o objeto será menor do que o calculado. A Figura 7.6 apresenta um exemplo de um objeto inserido. A posição aleatória escolhida foi o ponto A desta figura. O tamanho calculado do objeto foi ACFD, no entanto, a parte BCFE está fora do intervalo da matriz de nuvens de pontos. Sendo assim, o tamanho considerado foi ABED.

Adicionalmente, o número de objetos pode ser fixo para todos os trechos ou pode mudar aleatoriamente em cada trecho. Esta opção foi utilizada na implementação, conforme é ilustrado na Figura 7.7.

Se o conjunto NP' resultante da aplicação dessa estratégia for também constituído de np , np'_i é inserida logo após np_i , e o tamanho de T' é $2k$. Caso contrário, o tamanho de T' é k .

Figura 7.6: Exemplo de inserção de um obstáculo que ultrapasse a borda da matriz

Figura 7.7: Ilustração da geração de nuvens de pontos para a estratégia de inserção de obstáculos

7.2.6 Estratégia de combinação 6: Média (MED)

A estratégia de combinação Média é do tipo II e é definida da seguinte forma:

$MED(np_i(x1, y1, z1), np_{i+1}(x2, y2, z2)) \mapsto np'_m((x1 + x2)/2, (y1 + y2)/2, (z1 + z2)/2)$
para $i \in N, i = 1, \dots, k$ e $np_i, np_{i+1} \in T$.

Portanto, os eixos x, y, z da nova nuvem de pontos são calculados de acordo com a média dos eixos das duas nuvens de pontos que a precede e a sucede. Se o conjunto NP' resultante da aplicação dessa estratégia for também constituído de np , np'_m é inserida entre np_i e np_{i+1} e o tamanho de T' é $2k - 1$. Caso contrário, o tamanho de T é $k - 1$.

7.2.7 Estratégia de combinação 7: Troca de Metades Verticais (TMV)

Esta estratégia de combinação é do tipo III e é aplicada utilizando dois trechos T_i e T_{i+1} consecutivos de um log. A geração combina duas metades verticais de uma nuvem de pontos np_{ij} de T_i com a nuvem de pontos np_{i+1j} de T_{i+1} , gerando duas novas nuvens de pontos. Se o número de trechos de um log for ímpar, o último log é combinado com ele mesmo.

Se o conjunto NP' resultante da aplicação dessa estratégia for também constituído de np , np'_i é inserida logo após np_i , e o tamanho de T'_i e de T'_{i+1} é $2k$. A forma de intercalação e combinação são ilustradas na Figura 7.8. Caso contrário, o tamanho de T'_i e de T'_{i+1} é k . A forma de geração é ilustrada na Figura 7.9.

Figura 7.8: Esquema de combinação de pares de nuvens de pontos, incluindo as nuvens de pontos originais

Figura 7.9: Esquema de combinação de pares de nuvens de pontos, sem incluir as nuvens de pontos originais

7.2.8 Estratégia de combinação 8: Troca de Centrais (TC)

Esta estratégia de combinação também é do tipo III e é aplicada utilizando dois trechos T_i e T_{i+1} consecutivos de um log. A geração combina o centro de uma nuvem de pontos np_{i_j} de T_i com o centro da nuvem de pontos np_{i+1_j} de T_{i+1} , gerando duas novas nuvens de pontos. Se o número de trechos de um log for ímpar, o último log é combinado com ele mesmo. O tamanho do centro é definido por uma constante percentual p parametrizável, que especifica quantas colunas compõem o centro.

Se o conjunto NP' resultante da aplicação dessa estratégia for também constituído de np , np'_i é inserida logo após np_i , e o tamanho de T'_i e de T'_{i+1} é $2k$. As formas de intercalação e combinação são ilustradas na Figura 7.10. Caso contrário, o tamanho de T'_i e de T'_{i+1} é k . As regras de geração são ilustradas na Figura 7.11.

Figura 7.10: Esquema de combinação de trocas centrais, incluindo as nuvens de pontos originais

Figura 7.11: Esquema de combinação de trocas centrais, sem incluir as nuvens de pontos originais

7.3 Modelo e ferramenta PATeCa

Em relação ao modelo proposto no Capítulo 5, este capítulo trabalha principalmente com os conceitos *Estratégia* e *LogGerado* da Figura 7.12. O modelo considera que um *LogGerado* também pode ser obtido a partir de uma *Estratégia* que, por sua vez, pode ser do tipo Combinação e/ou Mutação e pode conter parâmetros específicos.

CAPÍTULO 7. ESTRATÉGIAS DE COMBINAÇÃO E MUTAÇÃO PARA GERAÇÃO DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

Figura 7.12: Modelo atualizado da PATECa

A ferramenta PATECa também foi estendida de forma a possibilitar a geração de novos logs utilizando estratégias de combinação e mutação a partir dos logs gerados pelas heurísticas, a simulação desses logs, o cálculo da cobertura e a comparação dessas coberturas tanto com os logs gerados a partir de heurísticas quanto com os logs gerados a partir de estratégias de combinação e mutação, calculando também a cobertura agregada. A Figura 7.13 apresenta a tela da ferramenta em que é possível escolher entre as estratégias de mutação e combinação gerar um log modificado de um log previamente selecionado.

Figura 7.13: Tela de geração de logs a partir de estratégias de combinação e mutação

A Figura 7.14 apresenta o diagrama de sequência para geração de logs, referente ao Caso de Uso “Gerar logs a partir de estratégias de combinação e mutação” da Figura 5.4. Primeiro, o testador seleciona um log a ser particionado e, após a geração desses trechos, o testador pode solicitar a geração de um novo log composto por trechos discriminatórios selecionados de acordo com a heurística informada ou por trechos selecionados manualmente por ele de maneira arbitrária. Em seguida, pode-se aplicar as estratégias de combinação e mutação para gerar um novo log e, então, realizar sua simulação a fim de calcular sua cobertura. Por meio da ferramenta, o testador pode visualizar o percentual de cobertura obtido para o critério todos-nós e todas-arestas das classes e métodos

CAPÍTULO 7. ESTRATÉGIAS DE COMBINAÇÃO E MUTAÇÃO PARA GERAÇÃO DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

envolvidos, o código fonte do programa destacando as linhas não executadas, o Grafo de Fluxo de Controle diferenciando os nós e arestas não executados dos executados. Também é possível selecionar dois logs para comparar a cobertura entre eles. Ao selecionar essa opção, a ferramenta também calcula o percentual de cobertura agregada para os critérios todos-nós e todas-arestas.

Figura 7.14: Diagrama de sequência para geração de logs

7.4 Estudo de coberturas obtidas com logs gerados a partir de estratégias de combinação e mutação

Um estudo de caso foi conduzido com dois objetivos: para validar a ferramenta desenvolvida e verificar a eficácia das estratégias de combinação e mutação definidas na Seção 7.2. Deve-se notar que essas estratégias podem ser implementadas com inúmeras variações de parâmetros. Para o estudo de caso foi utilizado o programa de desvio de obstáculos explicado na Seção 4.3 e os mesmos testes de campo utilizados no Capítulo 6. Após a realização dos testes de campo e da seleção dos nós discriminatórios descritos na Seção 6.3 foram gerados com a utilização da ferramenta novos logs compostos dos trechos discriminatórios selecionados pelas heurísticas H2 e H6 (as melhores de acordo com o estudo do Capítulo 6), para os métodos sob teste (*process* e *calule*). Esses logs foram utilizados como base para a aplicação das três estratégias de combinação e cinco estratégias de mutação definidas na Seção 7.2, gerando então novos logs.

O estudo de caso envolveu a simulação de quatorze (14) novos logs gerados pelas estratégias de mutação e combinação definidas na Seção 7.2 a partir dos cinco logs originais e das heurísticas H2 e H6. Para cada estratégia, foram calculadas as coberturas obtidas para os critérios todos-nós e todas-arestas dos métodos sob teste. Inicialmente foram realizados vários testes arbitrários com os valores dos parâmetros para definir quais produziam resultados melhores. Assim, as estratégias foram implementadas da seguinte forma:

CAPÍTULO 7. ESTRATÉGIAS DE COMBINAÇÃO E MUTAÇÃO PARA GERAÇÃO DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

- MCz10 foi implementada com o valor da constante cz igual a 10 e as demais como zero. O valor 10 ficou fixo. Na geração, optou-se por inserir as nuvens de pontos originais.
- MCx10 foi implementada com o valor da constante cx igual a 10 e as demais como zero. O valor 10 ficou fixo. Na geração, optou-se por inserir as nuvens de pontos originais.
- MCxz10: primeiramente MCx10 foi aplicada a T gerando T' e depois MCz10 foi aplicada a T' gerando T'' com o dobro do tamanho inicial. Tanto na geração de T' quanto de T'' , optou-se por inserir as nuvens de pontos originais.
- OBS1 foi implementado gerando aleatoriamente 1 ou 2 obstáculos de tamanhos e posições variados para cada nuvem de ponto. O comprimento e a largura de cada objeto foram calculados aleatoriamente e deveriam ter no máximo 3,3% de tl e 3,3% de tc , respectivamente. Na geração, optou-se por inserir as nuvens de pontos originais.
- OBS2 foi implementada gerando aleatoriamente 1 ou 2 obstáculos de tamanhos e posições variados para cada nuvem de ponto. Não houve restrição quanto ao tamanho dos objetos a serem inseridos. Na geração, optou-se por inserir as nuvens de pontos originais.
- RVH1: foi implementada e aplicada como definida. Na geração, optou-se por inserir as nuvens de pontos originais.
- RVH2: foi implementada e aplicada como definida. Na geração, as nuvens de pontos originais não foram inseridas.
- RV1: foi implementada e aplicada como definida. Na geração, optou-se por inserir as nuvens de pontos originais.
- RV2: foi implementada e aplicada como definida. Na geração, as nuvens de pontos originais não foram inseridas.
- RH1: foi implementada e aplicada como definida. Na geração, optou-se por inserir as nuvens de pontos originais.
- RH2: foi implementada e aplicada como definida. Na geração, as nuvens de pontos originais não foram inseridas.
- MED foi implementada e aplicada como definida. Na geração, optou-se por inserir as nuvens de pontos originais.

CAPÍTULO 7. ESTRATÉGIAS DE COMBINAÇÃO E MUTAÇÃO PARA GERAÇÃO DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

Tabela 7.1: Cobertura obtida para o critério todos-nós do método *process()* e heurística

H2																
Log	Log TC	Base	MED	MCz10	MCx10	MCxz10	OBS1	OBS2	RVH1	RVH2	RV1	RV2	RH1	RH2	TMV	TC
1	60%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%
2	71%	74%	74%	75%	75%	75%	74%	74%	75%	74%	75%	74%	74%	74%	74%	74%
3	70%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%
4	71%	74%	74%	75%	74%	75%	74%	75%	75%	74%	75%	74%	74%	74%	74%	74%
5	71%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%
média	68,6%	67,4%	67,4%	67,8%	67,6%	67,8%	67,4%	67,6%	67,8%	67,4%	67,8%	67,4%	67,4%	67,4%	67,4%	67,4%

Tabela 7.2: Cobertura obtida para o critério todos-nós do método *process()* e heurística

H6																
Log	Log TC	Base	MED	MCz10	MCx10	MCxz10	OBS1	OBS2	RVH1	RVH2	RV1	RV2	RH1	RH2	TMV	TC
1	60%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%
2	71%	74%	74%	74%	74%	75%	74%	75%	75%	74%	75%	74%	74%	74%	74%	74%
3	70%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%
4	71%	74%	74%	75%	75%	75%	74%	75%	75%	74%	75%	74%	74%	74%	74%	75%
5	71%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	63%	72%	63%
média	68,6%	67,4%	67,4%	67,6%	67,6%	67,8%	67,4%	67,8%	67,8%	67,4%	67,8%	67,4%	67,4%	67,4%	69,2%	67,6%

Tabela 7.3: Cobertura obtida para o critério todas-arestas do método *process()* e heurística H2

Log	Log TC	Base	MED	MCz10	MCx10	MCxz10	OBS1	OBS2	RVH1	RVH2	RV1	RV2	RH1	RH2	TMV	TC
1	46%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%
2	56%	58%	59%	60%	58%	60%	59%	59%	58%	58%	60%	58%	59%	58%	59%	59%
3	54%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%
4	56%	59%	59%	60%	59%	60%	59%	60%	60%	59%	60%	59%	59%	59%	59%	59%
5	56%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%
média	53,6%	52,2%	52,4%	52,8%	52,2%	52,8%	52,4%	52,6%	52,4%	52,2%	52,8%	52,2%	52,4%	52,2%	52,4%	52,4%

Tabela 7.4: Cobertura obtida para o critério todas-arestas do método *process()* e heurística H6

Log	Log TC	Base	MED	MCz10	MCx10	MCxz10	OBS1	OBS2	RVH1	RVH2	RV1	RV2	RH1	RH2	TMV	TC
1	46%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%
2	56%	58%	59%	59%	58%	58%	59%	60%	60%	58%	60%	58%	59%	58%	59%	59%
3	54%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%
4	56%	59%	59%	60%	60%	60%	59%	60%	60%	59%	60%	59%	59%	59%	59%	60%
5	56%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	48%	56%	48%
média	53,6%	52,2%	52,4%	52,6%	52,4%	52,4%	52,4%	52,8%	52,8%	52,2%	52,8%	52,2%	52,4%	52,2%	54%	52,6%

- TMV: foi implementada e aplicada como definida. Na geração, optou-se por inserir as nuvens de pontos originais.
- TC: foi implementada aplicando o valor de 70% à constante k. Na geração, optou-se por inserir as nuvens de pontos originais.

As Tabelas 7.1 e 7.2 apresentam as coberturas obtidas para o critério todos-nós do método *process()* para as heurísticas H2 e H6, respectivamente. Analogamente, as Tabelas 7.3 e 7.4 apresentam as coberturas obtidas para o critério todas-arestas do método *process()* para as heurísticas H2 e H6, respectivamente. Da mesma maneira, as Tabelas 7.5, 7.6, 7.7 e 7.8 apresentam as coberturas obtidas para os critérios todos-nós e todas-arestas do método *calcule()* para as heurísticas H2 e H6.

Pela análise das tabelas é possível observar que em alguns logs foi possível cobrir requisitos de teste que ainda não haviam sido cobertos nos logs base, embora o aumento da cobertura tenha sido pequeno. Esse resultado pode ser melhor visualizado nos gráficos das Figuras 7.15, 7.16, 7.17, 7.18, 7.19 e 7.20 que apresentam as médias das coberturas

CAPÍTULO 7. ESTRATÉGIAS DE COMBINAÇÃO E MUTAÇÃO PARA GERAÇÃO DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

Tabela 7.5: Cobertura obtida para o critério todos-nós do método *calcule()* e heurística

H2																
Log	Log TC	Base	MED	MCz10	MCx10	MCxz10	OBS1	OBS2	RVH1	RVH2	RV1	RV2	RH1	RH2	TMV	TC
1	91%	91%	91%	91%	91%	91%	91%	91%	91%	91%	91%	91%	91%	92%	91%	91%
2	93%	93%	92%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%
3	93%	91%	91%	91%	91%	91%	91%	91%	91%	92%	91%	91%	91%	91%	91%	91%
4	90%	90%	93%	90%	90%	90%	90%	90%	90%	90%	90%	90%	90%	90%	93%	92%
5	95%	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%
média	92,4%	91,4%	91,8%	91,4%	91,4%	91,4%	91,4%	91,4%	91,4%	91,6%	91,4%	91,4%	91,4%	91,6%	92%	91,8%

Tabela 7.6: Cobertura obtida para o critério todos-nós do método *calcule()* e heurística

H6																
Log	Log TC	Base	MED	MCz10	MCx10	MCxz10	OBS1	OBS2	RVH1	RVH2	RV1	RV2	RH1	RH2	TMV	TC
1	91%	91%	91%	91%	91%	91%	91%	91%	91%	91%	91%	91%	91%	92%	91%	92%
2	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%
3	93%	91%	91%	91%	91%	91%	91%	91%	91%	92%	91%	91%	91%	91%	91%	91%
4	90%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%
5	95%	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%	93%	92%
média	92,4%	92%	92%	92%	92%	92%	92%	92%	92%	92,2%	92%	92%	92%	92,2%	92,2%	92,20%

Tabela 7.7: Cobertura obtida para o critério todas-arestas do método *calcule()* e heurística H2

Log	Log TC	Base	MED	MCz10	MCx10	MCxz10	OBS1	OBS2	RVH1	RVH2	RV1	RV2	RH1	RH2	TMV	TC
1	74%	74%	74%	74%	74%	74%	74%	74%	74%	74%	74%	74%	74%	75%	74%	74%
2	78%	77%	77%	78%	77%	78%	78%	77%	78%	79%	78%	78%	77%	78%	77%	78%
3	77%	73%	73%	74%	74%	74%	73%	74%	73%	75%	74%	73%	73%	73%	73%	73%
4	74%	73%	77%	73%	73%	73%	73%	74%	73%	73%	73%	73%	74%	75%	77%	75%
5	81%	75%	75%	75%	75%	75%	75%	75%	75%	75%	75%	75%	75%	75%	75%	75%
média	76,8%	74,4%	75,2%	74,8%	74,6%	74,8%	74,6%	74,8%	74,6%	75,2%	74,8%	74,6%	74,6%	75,2%	75,2%	75%

Tabela 7.8: Cobertura obtida para o critério todas-arestas do método *calcule()* e heurística H6

Log	Log TC	Base	MED	MCz10	MCx10	MCxz10	OBS1	OBS2	RVH1	RVH2	RV1	RV2	RH1	RH2	TMV	TC
1	74%	74%	74%	74%	74%	74%	74%	74%	74%	74%	74%	74%	74%	75%	74%	75%
2	78%	78%	78%	79%	78%	79%	78%	78%	79%	79%	79%	79%	78%	78%	78%	78%
3	77%	73%	73%	74%	73%	74%	73%	73%	74%	75%	74%	73%	73%	73%	73%	73%
4	74%	77%	77%	77%	77%	77%	77%	77%	77%	77%	77%	77%	78%	78%	77%	77%
5	81%	75%	75%	75%	75%	75%	75%	75%	75%	75%	75%	75%	75%	75%	78%	75%
média	76,8%	75,4%	75,4%	75,8%	75,4%	75,8%	75,4%	75,4%	75,8%	76%	75,8%	75,6%	75,6%	75,8%	76%	75,6%

obtidas pelos cinco logs utilizados no estudo para as heurísticas H2 e H6, respectivamente. Pela média das coberturas, nota-se que as estratégias de mutação obtiveram resultados semelhantes, com uma leve vantagem para a estratégia de combinação TMV.

A Figura 7.21 apresenta gráficos que relacionam a cobertura obtida pelo log criado pelo teste de campo, pelo log base e pelos logs gerados utilizando as estratégias de mutação e combinação para o critério todos-nós (a) e para o critério todas-arestas (b) do método *process()*. Pela análise desses gráficos, pode-se observar que as estratégias obtiveram resultados muito semelhantes para os logs 1, 2, 3 e 4 em ambos os critérios, com nenhum ou leve aumento na cobertura. Para o log 5, a estratégia TVM conseguiu um aumento significativo na cobertura para a heurística H6. Já era esperado que as estratégias que envolvem a modificação dos valores das coordenadas x, y, z da nuvem de pontos não tivesse uma melhora da cobertura significativa para o método *process()* uma vez que nele não há nenhum comando de decisão que as envolva.

A Figura 7.22 apresenta gráficos semelhantes mas considerando o método *calcule()*. Para o critério todos-nós, o aumento da cobertura foi muito pequeno. No entanto, vale

CAPÍTULO 7. ESTRATÉGIAS DE COMBINAÇÃO E MUTAÇÃO PARA GERAÇÃO DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

Figura 7.15: Gráfico com as médias das coberturas obtidas para a heurística H2

Figura 7.16: Gráfico com as médias das coberturas obtidas para a heurística H2 do método *process()* para os critérios a) todos-nós e b) todas-arestas

CAPÍTULO 7. ESTRATÉGIAS DE COMBINAÇÃO E MUTAÇÃO PARA GERAÇÃO DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

Figura 7.17: Gráfico com as médias das coberturas obtidas para a heurística H2 do método *calcule()* para os critérios a) todos-nós e b) todas-arestas

Figura 7.18: Gráfico com as médias das coberturas obtidas para a heurística H6

CAPÍTULO 7. ESTRATÉGIAS DE COMBINAÇÃO E MUTAÇÃO PARA GERAÇÃO DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

Figura 7.19: Gráfico com as médias das coberturas obtidas para a heurística H6 do método *process()* para os critérios a) todos-nós e b) todas-arestas

Figura 7.20: Gráfico com as médias das coberturas obtidas para a heurística H6 do método *calcule()* para os critérios a) todos-nós e b) todas-arestas

CAPÍTULO 7. ESTRATÉGIAS DE COMBINAÇÃO E MUTAÇÃO PARA GERAÇÃO DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

Figura 7.21: Gráfico com as coberturas por log e heurística/estratégia do método *process()* para os critérios a) todos-nós e b) todas-arestas

CAPÍTULO 7. ESTRATÉGIAS DE COMBINAÇÃO E MUTAÇÃO PARA GERAÇÃO DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

ressaltar que o percentual coberto pelo log base já era muito alto (média de 91,4% para a heurística H2 e 92% para a heurística H6). Já para o critério todas-arestas, percebe-se um aumento da cobertura principalmente para as estratégias MCz10 e as de rotação (RVH1 a RH2).

É possível também observar que para a maioria dos logs do método *process()* e para alguns logs do método *calcule()*, a cobertura atingida para os critérios todos-nós e todas-arestas pelos logs base foi maior que a cobertura atingida pelo log criado no teste de campo. Considerando-se que a cobertura do método *calcule()* atingida pelo log criado no teste de campo já era alta, as chances de gerar novos logs que aumentassem a cobertura eram menores do que no método *process()*, que obteve uma cobertura razoavelmente menor para esse mesmo log.

Figura 7.22: Gráfico com as coberturas por log e heurística/estratégia do método *calcule()* para os critérios a) todos-nós e b) todas-arestas

A Tabela 7.9 apresenta quais estratégias obtiveram os melhores resultados para cada heurística e para cada método. A Tabela 7.10 apresenta o número de vezes em que

CAPÍTULO 7. ESTRATÉGIAS DE COMBINAÇÃO E MUTAÇÃO PARA GERAÇÃO DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

Tabela 7.9: Estratégias que obtiveram melhores resultados para cada heurística e para cada método

Método	H2 - Nós	H2 - Arestas	H6 - Nós	H6 - Arestas
process	MCz10	MCz10	MCxz10	OBS2
	MCxz10	MCxz10	OBS2	RVH1
	RVH1	RV1	RVH1	RV1
	RV1		RV1	TMV
calcule			TMV	
	MED	MED	RVH2	MCz10
	TMV	RVH2	RH2	MCxz10
	TC	RH2	TMV	RVH1
		TMV	TC	RVH2
				RV1
				RH2
				TMV

Tabela 7.10: Número de vezes em que cada estratégia de mutação foi melhor para cada heurística e para cada método

Estratégia	Número
RV1	5
MCxz10	4
RVH1	4
MCz10	3
RH2	3
RVH2	3
MED	2
OBS2	2

cada estratégia de mutação foi melhor para cada heurística e para cada método. Com isso, pode-se observar que as variações das estratégias de Rotação obtiveram os melhores resultados, pois são melhores em quase todas as heurísticas e métodos analisados. A estratégia de Rotação Vertical foi melhor em 5 dos 8 casos analisados. As estratégias de mudança de coordenadas também obtiveram bons resultados, conseguindo ser melhor em 4 dos 8 casos analisados para a estratégia MCxz10. A estratégia de Inserção de Obstáculos também aparece como uma das melhores para a heurística H6 do método *process()*, enquanto a estratégia de média aparece como uma das melhores para a heurística H2 do método *calcule()*.

7.5 Considerações finais

Neste capítulo foram definidas estratégias para combinação e mutação de logs, expandindo e detalhando as classes “*Estratégia*” e “*LogGerado*” do modelo apresentado no Capítulo 5. Também foi apresentada e discutida a implementação das estratégias na ferramenta PATeCa. Um estudo de caso envolvendo a aplicação dessas estratégias foi conduzido e notou-se que algumas delas produzem resultados melhores e que é possível aumentar a cobertura inicialmente obtida, ou seja, utilizando as estratégias de mutação e combinação é possível simular cenários não realizados durante a aplicação do log original de forma a conseguir um aumento na cobertura. No entanto, seria interessante gerar novos logs

CAPÍTULO 7. ESTRATÉGIAS DE COMBINAÇÃO E MUTAÇÃO PARA GERAÇÃO DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS

derivados a partir de estratégias que satisfazem uma função de adequação (*fitness*), como atingir um nó ou uma aresta específica, ou executar um dado comando com a finalidade de aumentar ainda mais a cobertura. Este assunto é tratado no capítulo seguinte.

Geração automática de dados de teste para softwares controladores de veículos autônomos utilizando algoritmos baseados em busca

8.1 Considerações iniciais

Algoritmos Baseados em Busca têm sido bastante utilizados na geração automática de dados de teste e, embora existam vários algoritmos conhecidos e validados para esse fim (McMinn, 2004), a maioria deles considera variáveis de entradas mais simples e comuns, como números e vetores. No entanto, em alguns domínios de aplicação há também a ocorrência de outros tipos mais complexos de estruturas de dados, como as nuvens de pontos para os veículos autônomos.

Nesse contexto, o objetivo deste capítulo é propor uma abordagem para geração de dados de teste utilizando algoritmos evolutivos e tendo como função objetivo (ou *fitness*) os requisitos do teste estrutural em geral e nesta tese, especificamente, os critérios todos-nós e todas-arestas. Nesta abordagem, a geração é realizada após a execução de um teste de campo e utiliza as entradas gravadas durante essa execução.

As propostas apresentadas neste capítulo correspondem ao caso de uso “Gerar logs usando algoritmo de busca” apresentado no diagrama de caso de uso da Figura 5.4 do Capítulo 5. A figura ilustra que para realizar essa atividade é necessário também usar as heurísticas definidas no Capítulo 6 e as estratégias definidas no Capítulo 7, bem como outras funcionalidades de execução, cálculo de cobertura, etc., implementadas pela ferramenta PATeCa e apresentadas genericamente no Capítulo 5. Com as novas classes e associações enfatizadas neste capítulo, o modelo de dados que vem sendo usado nesta tese é apresentado na Figura 8.1 da Seção 8.3 sem hachuras.

Este capítulo está organizado da seguinte forma: a Seção 8.2 apresenta um algoritmo genético para geração de dados de teste no contexto de veículos autônomos e que está baseado em entradas no formato de nuvens de pontos. A Seção 8.3 descreve sucintamente como esse algoritmo foi implementado na ferramenta PATeCa e discute algumas dificuldades encontradas. A Seção 8.4 ilustra o uso do algoritmo e apresenta um estudo com diferentes combinações de estratégias e heurísticas, analisando quais apresentam os melhores resultados. A Seção 8.5 apresenta as considerações finais do capítulo.

8.2 Algoritmo evolutivo para geração de dados de teste para software controlador de veículos autônomos

O algoritmo evolutivo apresentado na Seção 2.7.4 foi adaptado para o contexto de veículos autônomos que utilizam trechos que contêm nuvens de pontos como dados de entrada. O algoritmo adaptado é apresentado nos Algoritmos 3 e 4 em que os trechos são avaliados executando o programa sob teste por meio de simulação e utilizando-os como entrada. No final da execução do programa obtém-se o conjunto de dados de teste gerado (correspondente ao *LogGerado*) bem como uma lista dos requisitos de teste cobertos por eles e o percentual de cobertura. Assim como nos algoritmos genéticos tradicionais, a avaliação da função de *fitness* é utilizada para selecionar os pais da próxima geração e os operadores genéticos de combinação e mutação são utilizados para gerar a nova população de dados de teste. A avaliação da função de *fitness* depende do número de requisitos cobertos: ela é maior para o trecho que cobre o maior número de requisitos. Conforme explicado adiante, o cálculo da função de *fitness* é diferente para o conjunto utilizado como população inicial dos demais indivíduos gerados.

O algoritmo utiliza como entrada os seguintes dados: *versao*, que é a versão de um programa a ser testado; *logTC*, um log gerado a partir do teste de campo; *numNP*, que corresponde ao número de nuvens de pontos que cada trecho particionado de *logTC* deve conter; *fitness*, critério de teste estrutural a ser considerado: todos-nós ou todas-arestas; *tamPopulacao*, o tamanho da população de dados de teste a ser considerado na geração;

CAPÍTULO 8. GERAÇÃO AUTOMÁTICA DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS UTILIZANDO ALGORITMOS BASEADOS EM BUSCA

iteracoes, o número de iterações a ser considerado como um dos critérios de parada do algoritmo; *metodosSobTeste*: lista dos métodos da versão sob teste que deverão ser considerados na geração; *heuristica*, heurística a ser utilizada na seleção do conjunto inicial de trechos (*conjtoInicial*); *opCombinacao*, operador de combinação a ser utilizado na combinação dos pais para a geração de novos filhos; *opMutacao*, operador de mutação a ser utilizado para geração de novos filhos; e, *taxaMutacao*, valor da taxa de mutação a ser usado para gerar ou não um novo trecho mutado.

Adicionalmente, o algoritmo tem como variáveis locais: *trechosPart*, que armazena a lista de trechos particionado de *logTC*; *iteracaoAtual*, variável que armazena a quantidade de iterações que já foram realizadas; *mutacao*, valor em porcentagem calculado aleatoriamente para decidir se irá ou não gerar trechos mutados; *reqCobertos*, conjunto que armazena os requisitos que já foram cobertos pelos trechos gerados; *reqCobertosConjtoInicial*, conjunto que armazena os requisitos cobertos apenas por *conjtoInicial*; *populacaoCorrente*, lista que armazena os trechos referentes à população corrente; *conjtoInicial*, lista de trechos que foram selecionados pela heurística (*heuristica*) desejada; *filhos*, lista de trechos que foram gerados na iteração atual (*iteracaoAtual*); *cobertura*, percentual de cobertura atingido pela população.

O algoritmo começa inicializando e configurando as variáveis locais e de entrada (linhas 1-11). *logTC* é particionado em trechos contendo *numNP* nuvens de pontos cada e os trechos particionados são armazenados na lista *trechosPart*. Em seguida, são selecionados os trechos discriminatórios de acordo com *heuristica* para os métodos na lista *metodosSobTeste*. Esses trechos são armazenados na lista *conjtoInicial* e são os indivíduos da população inicial do algoritmo genético proposto.

A fim de obter uma lista de requisitos de teste de acordo com a função de *fitness*, o Grafo de Fluxo de Controle (GFC) é calculado (linha 3) e o número desses requisitos de teste é armazenado na variável *numRequisitos*. Para descobrir quais requisitos de teste foram cobertos para cada trecho, a *versao* do programa sob teste deve ser executada recebendo cada um dos trechos como entrada. Isso é realizado pelo procedimento *simular()*. Essa lista de requisitos cobertos é atribuída à variável *reqCobertosConjtoInicial*. Com isso, o valor da função objetivo de cada elemento de *conjtoInicial* é calculado segundo a fórmula: $numReqCobertosTrecho/numRequisitos$, em que *numReqCobertosTrecho* representa o número de requisitos cobertos por um dado trecho do conjunto inicial. Também na etapa de inicialização é calculada a cobertura inicial obtida por *conjtoInicial*, a lista de trechos de *conjtoInicial* é atribuída à variável *populacaoCorrente* e o valor zero é atribuído à variável *iteracaoAtual*.

O próximo passo é a geração de dados de teste propriamente dita (linhas 12-22). O laço **Enquanto** é executado até atingir 100% de cobertura ou até que o número de iterações

CAPÍTULO 8. GERAÇÃO AUTOMÁTICA DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS UTILIZANDO ALGORITMOS BASEADOS EM BUSCA

realizadas seja igual ao valor de *iteracoes*. O primeiro passo dessa fase é gerar os filhos da população corrente utilizando o operador de combinação informado em *opCombinacao* e atribuindo esse resultado à variável *filhos*. Em seguida, os filhos gerados são passados como parâmetros para o procedimento *realizaMutacao*. Nesse procedimento, para cada um dos indivíduos em *filhos* é gerado um valor aleatório *mutacao* a fim de compará-lo com a variável de entrada *taxaMutacao* com o objetivo de decidir se será aplicado ou não o operador de mutação para um dado indivíduo. Em caso positivo, o indivíduo será mutado, substituindo o indivíduo original na lista de *filhos*.

Como o número de trechos selecionados a partir da heurística (*conjtoInicial*) pode ser menor que o tamanho da população inicial informada em *tamPopulacao*, o algoritmo verifica se o número de trechos de *populacaoCorrente* + número de trechos gerados em *filhos* atingiu o tamanho desejado. Em caso positivo, o procedimento *selecionarPais* é invocado com a finalidade de descartar os indivíduos que menos contribuíram para a cobertura dos requisitos de teste, respeitando o valor de *tamPopulacao*; a variável *iteracaoAtual* é incrementada e o valor de cobertura atingido pelos trechos de *populacaoCorrente* é atualizada. Em caso negativo, os elementos de *filhos* são adicionados à lista de trechos de *populacaoCorrente* e o laço se repete. Nota-se, neste caso, que o valor da variável *iteracaoAtual* não sofre nenhuma alteração.

Para fazer a seleção dos indivíduos que permanecerão na próxima iteração, o procedimento *selecionarPais* primeiramente calcula os requisitos cobertos pelos trechos pertencentes à lista *filhos*, invocando o método *simular* para cada um deles. A etapa de simulação não é necessária para os indivíduos que estão na lista *populacaoCorrente* uma vez que já foi realizada em iterações anteriores. A partir da lista de requisitos cobertos por trecho, é possível calcular então a função de *fitness* para cada um deles segundo a fórmula: $numReqCobertosFilho / numRequisitos$, onde *numReqCobertosFilho* representa o número de requisitos cobertos por um dado trecho da lista *filhos*, desconsiderando os requisitos que já haviam sido cobertos pelos trechos em *conjtoInicial*. Vale ressaltar que esse cálculo é diferente do cálculo realizado para calcular a função *fitness* do conjunto inicial e funciona como um mecanismo de elitismo para dificultar que os trechos de *conjtoInicial* sejam descartados durante as iterações. Essa priorização é feita porque os trechos de *conjtoInicial* são indivíduos que foram selecionados porque possuíam grandes chances de aumentar a cobertura e, portanto, não seria vantajoso retirá-los da população. O próximo passo é adicionar os trechos de *filhos* à *populacaoCorrente* para que se possa ordenar de acordo com os valores da função de *fitness*. Com isso, os trechos que obtiveram os menores valores da função de *fitness* são removidos da *populacaoCorrente*, de tal forma que o número de trechos de *populacaoCorrente* seja igual a *tamPopulacao*.

CAPÍTULO 8. GERAÇÃO AUTOMÁTICA DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS UTILIZANDO ALGORITMOS BASEADOS EM BUSCA

Algorithm 3 Algoritmo Genético adaptado para gerar casos de testes para programas controladores de veículos autônomos

Entrada:

fitness: critério de cobertura a ser utilizado: nós ou arestas
heuristica: heurística a ser considerada na seleção do conjunto de trechos inicial
iteracoes: número de iterações a ser considerada como critério de parada
logTC: log gerado a partir do teste de campo
metodosSobTeste: lista de métodos a serem considerados na função de fitness
numNP: número de nuvens de pontos em cada trecho
opCombinacao: operador de combinação a ser utilizado
opMutacao: operador de mutação a ser utilizado
tamPopulacao: tamanho da população
tazaMutacao: valor da taxa de mutação
versao: versão do programa sob teste

Variáveis:

cobertura: valor de cobertura
conjtoInicial: lista contendo os trechos discriminatórios de *logTC*
filhos: lista contendo os indivíduos gerados por combinação na *iteracaoAtual*
iteracaoAtual: número de iterações que já foram realizadas
mutacao: valor da taxa de mutação calculada aleatoriamente
numRequisitos: número de requisitos de teste
populacaoCorrente, *conjtoInicial*, *filhos*: listas de trechos
reqCobertos: conjunto dos requisitos de teste cobertos pelos dados gerados
reqCobertosConjtoInicial: conjunto dos requisitos de teste que foram cobertos pelos trechos em *conjtoInicial*
trechosPart: lista contendo trechos de *logTC* particionado

Funções e procedimentos:

gerarConjuntoInicial(): função que retorna uma lista com os trechos discriminatórios de *logTC*
particionarLog(): procedimento responsável por particionar *logTC* em trechos contendo *numNP* nuvens de pontos
simular(): procedimento que executa a *versao* do programa sob teste tendo como entrada um determinado trecho
combinarPopulacao(): procedimento que gera novos indivíduos combinando-os dois a dois de acordo com *opCombinacao*
realizarMutacao(): procedimento que calcula um valor aleatório para cada indivíduo e realiza a mutação se esse valor for maior ou igual que *tazaMutacao*
selecionarPais(): procedimento que faz a seleção dos pais para a próxima iteração

/**** Inicialização e configuração ****/

- 1: *trechosPart* $\leftarrow$ *particionarLog*(*numNP*) (Ver Seção 6.2)
- 2: *conjtoInicial* $\leftarrow$ *gerarConjuntoInicial*(*trechosPart*, *heuristica*, *metodosSobTeste*)
- 3: Calcular o GFC dos métodos sob teste
- 4: Calcular requisitos de teste e atribuir a quantidade de requisitos à variável *numRequisitos*
- 5: *simular*(*conjtoInicial*) (Ver Seção 5.2)
- 6: Computar os requisitos cobertos de *conjtoInicial* e adicioná-los em *reqCobertos*
- 7: Computar o valor de fitness dos elementos de *conjtoInicial*, segundo a fórmula $\frac{\text{numReqCobertosTrecho}}{\text{numRequisitos}}$
- 8: Armazenar os requisitos cobertos por *conjtoInicial* em *requisitosCobCnjtoInicial*
- 9: *cobertura* $\leftarrow$ *reqCobertos.size()*/*numRequisitos*
- 10: *populacaoCorrente* $\leftarrow$ *conjtoInicial*
- 11: *iteracaoAtual* $\leftarrow$ 0

/**** Geração dos dados de teste ****/

- 12: **Enquanto** *iteracaoAtual* < *iteracoes* **E** *cobertura* != 1 **faça**
 - 13: *filhos* $\leftarrow$ *combinarPopulacao*(*populacaoCorrente*, *opCombinacao*) (Ver Seção 7.2)
 - 14: *filhos* $\leftarrow$ *realizarMutacao*(*filhos*, *opMutacao*, *tazaMutacao*) (Ver Seção 7.2)
 - 15: **Se** (*filhos.size()* + *populacaoCorrente.size()* > *tamPopulacao*) **então**
 - 16: *populacaoCorrente* $\leftarrow$ *selecionarPais*(*populacaoCorrente*, *filhos*)
 - 17: *iteracaoAtual*++
 - 18: *cobertura* = *reqCobertos.size()*/*numRequisitos*
 - 19: **Senão**
 - 20: *populacaoCorrente* $\leftarrow$ *populacaoCorrente* + *filhos*
 - 21: **Fim Se**
 - 22: **Fim Enquanto**
-

CAPÍTULO 8. GERAÇÃO AUTOMÁTICA DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS UTILIZANDO ALGORITMOS BASEADOS EM BUSCA

Algorithm 4 Algoritmo Genético adaptado para gerar casos de testes para programas controladores de veículos autônomos (continuação)

```

23: Função COMBINARPOPULACAO(populacao, opCombinacao)
24:   pai ← nulo
25:   mae ← nulo
26:   Para p em populacao faça
27:     Se pai = nulo então
28:       pai ← p
29:       continua
30:     Senão
31:       mae ← p
32:       filho ← Combinar pai e mae aplicando opCombinacao
33:       filhos ← filhos + filho
34:       pai ← nulo
35:       mae ← nulo
36:     Fim Se
37:   Fim Para
38:   Se pai != nulo então
39:     filho ← Combinar pai e pai aplicando opCombinacao
40:     filhos ← filhos + filho
41:   Fim Se
42:   Retorna filhos
43: Fim Função

44: Procedimento REALIZARMUTACAO(filhos, opMutacao, taxaMutacao)
45:   Para filho em filhos faça
46:     mutacao ← random(1..100)
47:     Se mutacao ≤ taxaMutacao então
48:       filho ← Mutar filho aplicando opMutacao
49:     Fim Se
50:   Fim Para
51: Fim Procedimento

52: Procedimento SELECIONARPAIS(populacaoCorrente, filhos)
53:   simular(filhos)
54:   Para filho em filhos faça
55:     filho.reqCobertos ← filho.reqCobertos - reqCobertosConjtoInicial
56:     fitness ← filho.reqCobertos.size() / numRequisitos
57:     populacaoCorrente ← populacaoCorrente + filhos
58:   Ordenar populacaoCorrente de maneira decrescente, considerando a função de fitness de cada elemento
59:   Retirar do final da lista os (populacaoCorrente.size() - tamPopulacao) indivíduos de populacaoCorrente
60:   Fim Para
61: Fim Procedimento

```

Uma segunda versão do algoritmo foi implementada para que a cada iteração fosse escolhido um operador de combinação e mutação aleatoriamente, como pode ser visto no Algoritmo 5.

8.3 Modelo e ferramenta PATeCa

Em relação ao modelo proposto no Capítulo 5, este capítulo desenvolve principalmente os conceitos *GeraçãoDT* e *AlgoritmoGenético* da Figura 8.1. *GeraçãoDT* é uma classe genérica para geração de dados de teste utilizando algoritmos de busca e que utiliza *Heurística* para seleção da população inicial e *Estratégia* como operadores de combinação e mutação. *AlgoritmoGenético* é uma instância de *GeraçãoDT* que implementa um algoritmo genético. No entanto, o modelo permite que outros algoritmos baseados em busca possam ser

CAPÍTULO 8. GERAÇÃO AUTOMÁTICA DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS UTILIZANDO ALGORITMOS BASEADOS EM BUSCA

Algorithm 5 Versão de escolha aleatória das estratégias para os procedimentos *combinarPopulacao* e *realizarMutacao*

```

Função COMBINARPOPULACAO(populacao)
  opCombinacao  $\leftarrow$  random(TMV, TC)
  pai  $\leftarrow$  nulo
  mae  $\leftarrow$  nulo
  Para p em populacao faça
    ...
  Fim Para
  Se pai  $\neq$  nulo então
    filho  $\leftarrow$  Combinar pai e pai aplicando opCombinacao
    filhos  $\leftarrow$  filhos + filho
  Fim Se
  Retorna filhos
Fim Função

Procedimento REALIZARMUTACAO(filhos, taxaMutacao)
  opMutacao  $\leftarrow$  random(MED, MC, OBS, RVH, RV, RH)
  Para filho em filhos faça
    mutacao  $\leftarrow$  random(1..100)
    Se mutacao  $\leq$  taxaMutacao então
      filho  $\leftarrow$  Mutar filho aplicando opMutacao
    Fim Se
  Fim Para
Fim Procedimento

```

acrescentados no futuro como, por exemplo, *Particle Swarm Optimization*. *GeraçãoDT* é uma classe abstrata que representa os elementos comuns desse tipo de algoritmo.

O algoritmo foi implementado na ferramenta PATeCa a fim de oferecer suporte à abordagem proposta. A Figura 8.2 apresenta a tela para geração de dados de teste. Nessa tela, o usuário deverá informar o diretório onde os logs gerados serão armazenados, o nome do novo log gerado e os demais parâmetros que são utilizados no algoritmo genético, tais como: tamanho da população, taxa de mutação, a função de *fitness*, o número de iterações, as estratégias de combinação e mutação utilizadas e os métodos sob teste. O usuário também poderá optar para que as estratégias de combinação e mutação sejam escolhidas de forma aleatória no algoritmo. Ao final do algoritmo, a ferramenta salva o estado dos objetos utilizados, tornando possível retomar essa geração mais tarde com um número de iterações maior. Por exemplo, uma geração pode ser iniciada com um número de 10 iterações e, ao final, o algoritmo retornará o conjunto de logs gerados e o valor da cobertura alcançada por eles. Caso o usuário não esteja satisfeito, poderá tentar aumentar a cobertura ainda mais informando um número maior de iterações e qual é a iteração inicial. No caso do exemplo com 10 iterações, a ferramenta iniciará a partir da décima primeira iteração. Essa funcionalidade foi implementada na ferramenta devido ao longo tempo de execução do algoritmo.

A Figura 8.3 apresenta a tela de Gerenciamento de Logs, na qual é possível também comparar as coberturas de todos os logs gerados, seja por meio das heurísticas, pela utilização apenas das estratégias de combinação e mutação ou pelo log gerado utilizando algoritmos de busca. A Figura 8.4 a), por exemplo, apresenta a comparação da cobertura obtida por dois logs gerados por meio de seus respectivos grafos de fluxo de controle.

CAPÍTULO 8. GERAÇÃO AUTOMÁTICA DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS UTILIZANDO ALGORITMOS BASEADOS EM BUSCA

Figura 8.1: Modelo completo

Figura 8.2: Tela de geração automática da PATECa

CAPÍTULO 8. GERAÇÃO AUTOMÁTICA DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS UTILIZANDO ALGORITMOS BASEADOS EM BUSCA

A Figura 8.4 b) apresenta o grafo agregado resultante em que é possível notar na área circulada que todos os nós aparecem como cobertos (pintados de verde), uma vez que esses mesmos nós foram cobertos por pelo menos um dos logs considerados.

Figura 8.3: Tela de gerenciamento dos logs

Figura 8.4: a) Comparação dos grafos de cobertura de dois logs gerados e b) grafo agregado resultante

8.3.1 Dificuldades encontradas

Em relação à implementação da ferramenta, algumas dificuldades foram encontradas. O cálculo da cobertura é baseada na ferramenta GCov que, após a execução do programa, gera os arquivos **.gcda* contendo as informações de cobertura obtida (ver Seção 5.3). No entanto, para algumas funcionalidades da ferramenta era necessário que esses arquivos fossem gerados após a simulação do log, com o programa ainda em execução. Para que isso fosse possível, o código-fonte da versão do programa sob teste foi modificado para reconhecer o sinal *SIGUSR1* e chamar a função *_gcov_flush()*.

A fim de aumentar o desempenho da geração automática de logs utilizando algoritmos genéticos, o cálculo da cobertura dos indivíduos gerados foi implementado utilizando *threads*, isto é, a cobertura dos indivíduos é realizada em paralelo. O módulo do cálculo da cobertura utiliza as ferramentas do ROS para simular o programa que, por sua vez, utiliza uma arquitetura *publisher-subscriber*, fazendo *broadcast* das mensagens (entradas) recebidas. Foi necessário recompilar a versão do programa a cada nova simulação, renomeando as mensagens recebidas tanto para cada versão do programa quanto para o log gerado, para fazer os cálculos corretamente. Sem esse ajuste, todos os programas e todos os logs que estivessem sendo executados e reproduzidos, respectivamente, em paralelo estariam recebendo e enviando, respectivamente, mensagens com o mesmo ID. Ou seja, todas as simulações em paralelo receberiam todas as mensagens dos logs que estavam sendo reproduzidos no momento.

8.4 Validação da abordagem proposta

Para validar o algoritmo proposto, foram utilizados os mesmos logs obtidos a partir dos cinco testes de campo descritos na Seção 6.2. Os cinco logs foram utilizados como entrada da variável *logTC* para aplicação do Algoritmo 3. Optou-se por escolher os conjuntos gerados pelos trechos contendo cinco nuvens de pontos por conter um número menor de variáveis de entradas e conseguir uma cobertura parecida com a dos trechos contendo dez nuvens de pontos. Além disso, as heurísticas H2 e H6 foram escolhidas por terem obtido os melhores resultados no estudo de caso discutido na Seção 6.3. A heurística H7 também foi incluída para ser utilizada como controle e analisar se vale a pena utilizar uma população inicial que já possuía boa cobertura.

O tamanho da população escolhido foi 25, resultando em um conjunto gerado com 125 nuvens de pontos, e foram realizadas 10 iterações. O número pequeno de iterações deve-se ao fato do longo tempo de execução do algoritmo e também pela ferramenta possuir a possibilidade de aumentar esse número incrementalmente. A Tabela 8.1 apresenta as entradas utilizadas em cada execução do algoritmo e o tempo gasto em cada uma dessas execuções. Nessa tabela, a coluna *ID* indica o ID da execução; a coluna *Log* indica qual o log que foi executado; a coluna *Tam. Pop.* indica o tamanho da população utilizado como entrada; a coluna *Iter* indica o número de iterações realizadas; a coluna *MetodosSobTeste* lista quais os métodos sob teste que foram considerados; a coluna *Heurist.* indica a heurística considerada; a coluna *Alg.* indica qual algoritmo foi utilizado (lembrando que as estratégias de combinação e mutação eram informadas como parâmetro no Algoritmo 3 e selecionadas aleatoriamente no Algoritmo 5); a coluna *Est. Mut.* indica qual estratégia de mutação foi informada como parâmetro no Algoritmo 3; a coluna *Tx. Mut.* indica o

CAPÍTULO 8. GERAÇÃO AUTOMÁTICA DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS UTILIZANDO ALGORITMOS BASEADOS EM BUSCA

percentual utilizado para o parâmetro *taxaMutacao* do algoritmo; e a coluna *Tempo Exec.* indica o tempo gasto pelo algoritmo para ser executado. Ao todo foram realizadas 80 execuções, sendo 60 para o Algoritmo 3, em que a estratégia de combinação TMV era fixa, variando-se as estratégias de mutação OBS, MC e RV, conforme apresentado nessa tabela. Essas estratégias de mutação foram escolhidas por se tratarem das estratégias que obtiveram os melhores resultados no estudo de caso discutido na Seção 7.4. Em todas as execuções, a taxa de mutação utilizada foi de 20%.

ID	Log	Tam. Pop.	Iter.	MetodosSobTeste	Heurist.	Alg.	Est. Mut.	Tx. Mut.	Tempo Exec.
1	1	25	10	<calcule>	H2	5	-	20%	53M53S
2	1	25	10	<calcule>	H2	3	MC	20%	52M21S
3	1	25	10	<calcule>	H2	3	OBS	20%	52M23S
4	1	25	10	<calcule>	H2	3	RV	20%	53M19S
5	1	25	10	<process>	H2	5	-	20%	50M49S
6	1	25	10	<process>	H2	3	MC	20%	51M28S
7	1	25	10	<process>	H2	3	OBS	20%	50M22S
8	1	25	10	<process>	H2	3	RV	20%	50M54S
9	1	25	10	<process, calcule>	H6	5	-	20%	49M35S
10	1	25	10	<process, calcule>	H6	3	MC	20%	53M54S
11	1	25	10	<process, calcule>	H6	3	OBS	20%	50M1S
12	1	25	10	<process, calcule>	H6	3	RV	20%	50M1S
13	1	25	10	<process, calcule>	H7	5	-	20%	53M8S
14	1	25	10	<process, calcule>	H7	3	MC	20%	52M28S
15	1	25	10	<process, calcule>	H7	3	OBS	20%	52M11S
16	1	25	10	<process, calcule>	H7	3	RV	20%	53M6S
17	2	25	10	<calcule>	H2	5	-	20%	52M14S
18	2	25	10	<calcule>	H2	3	MC	20%	52M57S
19	2	25	10	<calcule>	H2	3	OBS	20%	52M13S
20	2	25	10	<calcule>	H2	3	RV	20%	52M57S
21	2	25	10	<process>	H2	5	-	20%	52M25S
22	2	25	10	<process>	H2	3	MC	20%	53M7S
23	2	25	10	<process>	H2	3	OBS	20%	53M28S
24	2	25	10	<process>	H2	3	RV	20%	53M33S
25	2	25	10	<process, calcule>	H6	5	-	20%	53M8S
26	2	25	10	<process, calcule>	H6	3	MC	20%	52M33S
27	2	25	10	<process, calcule>	H6	3	OBS	20%	52M43S
28	2	25	10	<process, calcule>	H6	3	RV	20%	52M50S
29	2	25	10	<process, calcule>	H7	5	-	20%	52M45S
30	2	25	10	<process, calcule>	H7	3	MC	20%	52M45S
31	2	25	10	<process, calcule>	H7	3	OBS	20%	52M57S
32	2	25	10	<process, calcule>	H7	3	RV	20%	53M19S
33	3	25	10	<calcule>	H2	5	-	20%	52M37S
34	3	25	10	<calcule>	H2	3	MC	20%	52M43S
35	3	25	10	<calcule>	H2	3	OBS	20%	52M39S
36	3	25	10	<calcule>	H2	3	RV	20%	52M29S
37	3	25	10	<process>	H2	5	-	20%	52M49S
38	3	25	10	<process>	H2	3	MC	20%	53M14S
39	3	25	10	<process>	H2	3	OBS	20%	53M18S
40	3	25	10	<process>	H2	3	RV	20%	53M8S
41	3	25	10	<process, calcule>	H6	5	-	20%	53M43S
42	3	25	10	<process, calcule>	H6	3	MC	20%	54M9S
43	3	25	10	<process, calcule>	H6	3	OBS	20%	52M47S

CAPÍTULO 8. GERAÇÃO AUTOMÁTICA DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS UTILIZANDO ALGORITMOS BASEADOS EM BUSCA

44	3	25	10	<process, calcule>	H6	3	RV	20%	53M35S
45	3	25	10	<process, calcule>	H7	5	-	20%	52M43S
46	3	25	10	<process, calcule>	H7	3	MC	20%	52M57S
47	3	25	10	<process, calcule>	H7	3	OBS	20%	53M20S
48	3	25	10	<process, calcule>	H7	3	RV	20%	52M57S
49	4	25	10	<calcule>	H2	5	-	20%	52M21S
50	4	25	10	<calcule>	H2	3	MC	20%	52M50S
51	4	25	10	<calcule>	H2	3	OBS	20%	52M46S
52	4	25	10	<calcule>	H2	3	RV	20%	52M11S
53	4	25	10	<process>	H2	5	-	20%	53M28S
54	4	25	10	<process>	H2	3	MC	20%	53M35S
55	4	25	10	<process>	H2	3	OBS	20%	53M14S
56	4	25	10	<process>	H2	3	RV	20%	53M9S
57	4	25	10	<process, calcule>	H6	5	-	20%	53M3S
58	4	25	10	<process, calcule>	H6	3	MC	20%	53M37S
59	4	25	10	<process, calcule>	H6	3	OBS	20%	53M9S
60	4	25	10	<process, calcule>	H6	3	RV	20%	53M1S
61	4	25	10	<process, calcule>	H7	5	-	20%	53M5S
62	4	25	10	<process, calcule>	H7	3	MC	20%	52M49S
63	4	25	10	<process, calcule>	H7	3	OBS	20%	52M45S
64	4	25	10	<process, calcule>	H7	3	RV	20%	52M34S
65	5	25	10	<calcule>	H2	5	-	20%	53M16S
66	5	25	10	<calcule>	H2	3	MC	20%	54M10S
67	5	25	10	<calcule>	H2	3	OBS	20%	54M24S
68	5	25	10	<calcule>	H2	3	RV	20%	53M12S
69	5	25	10	<process>	H2	5	-	20%	50M21S
70	5	25	10	<process>	H2	3	MC	20%	51M35S
71	5	25	10	<process>	H2	3	OBS	20%	51M22S
72	5	25	10	<process>	H2	3	RV	20%	50M56S
73	5	25	10	<process, calcule>	H6	5	-	20%	49M31S
74	5	25	10	<process, calcule>	H6	3	MC	20%	50M42S
75	5	25	10	<process, calcule>	H6	3	OBS	20%	50M43S
76	5	25	10	<process, calcule>	H6	3	RV	20%	49M47S
77	5	25	10	<process, calcule>	H7	5	-	20%	53M5S
78	5	25	10	<process, calcule>	H7	3	MC	20%	53M5S
79	5	25	10	<process, calcule>	H7	3	OBS	20%	53M37S
80	5	25	10	<process, calcule>	H7	3	RV	20%	53M14S
Média:									52M34S
Total:									70H05M32S

Tabela 8.1: Variáveis de entrada e tempo de cada execução

Ressalta-se que tanto nas estratégias de combinação quanto nas estratégias de mutação, as nuvens de pontos originais não foram inseridas nos novos logs gerados pois isso aumentaria o número de nuvens de pontos exponencialmente a cada iteração, inviabilizando a geração. Além disso, os parâmetros requeridos por algumas das estratégias eram calculados de maneira aleatória pela ferramenta a cada iteração. Por exemplo, para a execução de ID 3 da Tabela 8.1, cuja estratégia de mutação utilizada foi OBS, os trechos gerados na iteração 1 poderiam ter utilizado como parâmetros: 20 para a coordenada Z, 10 para o número máximo de obstáculos e 40% para o tamanho máximo dos obstáculos;

CAPÍTULO 8. GERAÇÃO AUTOMÁTICA DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS UTILIZANDO ALGORITMOS BASEADOS EM BUSCA

Tabela 8.2: Cobertura obtida para o critério todos-nós do método *process*

Log	Base	RV e TMV	OBS e TMV	Aleat.	MC e TMV	Base	RV e TVM	OBS e TMV	Aleat.	MC e TMV	Base	RV e TMV	OBS e TMV	Aleat.	MC e TMV
	H2	H2	H2	H2	H2	H6	H6	H6	H6	H6	H7	H7	H7	H7	H7
1	63%	71%	71%	70%	70%	63%	71%	71%	70%	70%	61%	61%	61%	61%	61%
2	74%	79%	79%	79%	79%	74%	79%	79%	78%	79%	70%	70%	73%	72%	70%
3	63%	70%	70%	70%	70%	63%	70%	70%	70%	70%	60%	60%	60%	60%	60%
4	74%	79%	79%	79%	79%	74%	79%	79%	79%	79%	61%	61%	61%	61%	61%
5	63%	71%	71%	70%	70%	63%	70%	70%	70%	70%	61%	61%	61%	61%	61%
média	67%	74%	74%	74%	74%	67%	74%	74%	73%	74%	63%	63%	63%	63%	63%

Tabela 8.3: Cobertura obtida para o critério todas-arestas do método *process*

Log	Base	RV e TMV	OBS e TMV	Aleat.	MC e TMV	Base	RV e TVM	OBS e TMV	Aleat.	MC e TMV	Base	RV e TMV	OBS e TMV	Aleat.	MC e TMV
	H2	H2	H2	H2	H2	H6	H6	H6	H6	H6	H7	H7	H7	H7	H7
1	48%	59%	59%	59%	59%	48%	59%	59%	59%	59%	48%	48%	48%	48%	48%
2	58%	72%	71%	72%	72%	58%	72%	72%	66%	72%	56%	56%	59%	57%	56%
3	48%	59%	59%	59%	59%	48%	59%	59%	59%	59%	48%	48%	48%	48%	48%
4	59%	72%	72%	72%	72%	59%	72%	72%	72%	72%	48%	48%	48%	48%	48%
5	48%	59%	59%	59%	59%	48%	59%	59%	59%	59%	48%	48%	48%	48%	48%
média	52%	64%	64%	64%	64%	52%	64%	64%	63%	64%	50%	50%	50%	50%	50%

enquanto na iteração 2, os parâmetros: -10 para a coordenada Z, 2 para o número máximo de obstáculos e 80% para o tamanho máximo dos obstáculos.

Para o Algoritmo 5 foram realizadas 20 execuções e, a cada iteração dessas execuções, tanto as estratégias de combinação quanto as de mutação eram escolhidas aleatoriamente. Por exemplo, para a execução de ID 9, na iteração 1 foi escolhido a estratégia de combinação TMV e a estratégia RH para mutação e, assim, todos os novos trechos dessa iteração foram gerados aplicando essas estratégias. Na segunda iteração, as estratégias de combinação e mutação escolhidas aleatoriamente foram TC e MED, respectivamente, e de forma análoga a primeira iteração, todos os trechos foram gerados aplicando TC e MED, e assim sucessivamente para as demais iterações. Cada execução levou em média 52min para ser concluída e o tempo total gasto das execuções foi de 70 horas.

Após o fim de cada execução do algoritmo, o conjunto dos trechos gerados foi simulado a fim de computar a cobertura obtida. Esses resultados podem ser vistos nas Tabelas 8.2, 8.3, 8.4 e 8.5. Nessas tabelas, a coluna *Aleat.* indica que o Algoritmo 5, que determina as estratégias de combinação e mutação aleatoriamente, foi utilizado. Comparando o valor da cobertura obtida por meio das técnicas de geração com a cobertura do log base, nota-se que houve uma boa melhora na cobertura. Para o log 2, por exemplo, a cobertura para o critério todas-arestas passou de 52% para 72%. O resultado foi satisfatório, uma vez que tanto o número de iterações quanto o tamanho da população utilizados foram pequenos. Além disso, para alguns logs a cobertura já estava muito alta, o que torna mais difícil encontrar novos dados de teste que cubram os requisitos ainda não cobertos.

Os gráficos das Figuras 8.5 e 8.6 apresentam, respectivamente, as médias das coberturas obtidas para o critério todos-nós e todas-arestas do método *process*. Nota-se que neste caso não houve diferença de cobertura entre as heurísticas H2 e H6 e as estratégias de mutação utilizadas. Foi realizada uma análise do código fonte e percebeu-se que os critérios de teste ainda não cobertos pelos logs que obtiveram maior cobertura não depen-

CAPÍTULO 8. GERAÇÃO AUTOMÁTICA DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS UTILIZANDO ALGORITMOS BASEADOS EM BUSCA

Tabela 8.4: Cobertura obtida para o critério todos-nós do método *calcule*

Log	Base	RV e TMV	OBS e TMV	Aleat. e TMV	MC e TMV	Base	RV e TVM	OBS e TMV	Aleat. e TMV	MC e TMV	Base	RV e TMV	OBS e TMV	Aleat. e TMV	MC e TMV
	H2	H2	H2	H2	H2	H6	H6	H6	H6	H6	H7	H7	H7	H7	H7
1	91%	91%	91%	91%	91%	92%	92%	92%	92%	92%	86%	89%	92%	92%	89%
2	93%	93%	93%	93%	93%	93%	93%	93%	93%	93%	90%	93%	93%	93%	90%
3	91%	91%	91%	91%	91%	91%	91%	91%	91%	91%	83%	86%	86%	86%	86%
4	90%	93%	90%	90%	90%	93%	93%	93%	93%	93%	83%	86%	86%	86%	86%
5	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%	90%	90%	90%	90%	90%
média	91%	92%	91%	91%	91%	92%	92%	92%	92%	92%	86%	89%	89%	89%	88%

Tabela 8.5: Cobertura obtida para o critério todas-arestas do método *calcule*

Log	Base	RV e TMV	OBS e TMV	Aleat. e TMV	MC e TMV	Base	RV e TVM	OBS e TMV	Aleat. e TMV	MC e TMV	Base	RV e TMV	OBS e TMV	Aleat. e TMV	MC e TMV
	H2	H2	H2	H2	H2	H6	H6	H6	H6	H6	H7	H7	H7	H7	H7
1	74%	74%	74%	74%	74%	75%	75%	75%	75%	75%	67%	72%	75%	75%	72%
2	77%	77%	77%	77%	77%	78%	78%	78%	79%	78%	72%	76%	76%	77%	72%
3	73%	74%	73%	73%	74%	73%	74%	74%	74%	74%	64%	69%	69%	69%	69%
4	73%	77%	73%	73%	73%	77%	77%	77%	77%	77%	63%	68%	68%	68%	68%
5	75%	75%	75%	75%	75%	75%	75%	75%	75%	75%	73%	73%	73%	73%	73%
média	74%	75%	74%	74%	75%	76%	76%	76%	76%	76%	68%	72%	72%	72%	71%

diam da nuvem de pontos. Ou seja, não seria mais possível criar novos logs que cobrissem esses requisitos que ficaram faltando e, sendo assim, não foram realizadas novas iterações. A heurística H7 teve o pior desempenho, confirmando a hipótese de que com um bom conjunto de população inicial obtém-se melhores resultados.

Figura 8.5: Gráfico com as médias das coberturas obtidas para o critério todos-nós do método *process*

Os gráficos das Figuras 8.7 e 8.8 apresentam, respectivamente, as médias das coberturas obtidas para o critério todos-nós e todas-arestas do método *calcule*. Nota-se neste caso que, em geral, a heurística H6 obteve os melhores resultados. Para a heurística H2, a estratégia de mutação RV obteve o melhor resultado para o Algoritmo 3. Os novos logs gerados a partir da heurística H7 também obtiveram melhora na cobertura em relação a sua cobertura inicial. No entanto, nenhuma das estratégias conseguiu atingir a mesma cobertura alcançada pelos logs gerados a partir das heurísticas H2 e H6. Para a heurística H7, a estratégia de mutação MC foi levemente pior em relação às outras utilizadas que obtiveram a mesma cobertura.

Em resumo, as estratégias de mutação tiveram resultados semelhantes, exceto a estratégia RV, que teve uma pequena vantagem sobre as demais. No entanto, vale ressaltar

CAPÍTULO 8. GERAÇÃO AUTOMÁTICA DE DADOS DE TESTE PARA SOFTWARES CONTROLADORES DE VEÍCULOS AUTÔNOMOS UTILIZANDO ALGORITMOS BASEADOS EM BUSCA

Figura 8.6: Gráfico com as médias das coberturas obtidas para o critério todas-arestas do método *process*

Figura 8.7: Gráfico com as médias das coberturas obtidas para o critério todos-nós do método *calcule*

Figura 8.8: Gráfico com as médias das coberturas obtidas para o critério todas-arestas do método *calcule*

que essa estratégia não utiliza parâmetros e talvez o pior resultado das demais tenha sido graças a escolhas ruins dos parâmetros que foram calculados aleatoriamente.

8.5 Considerações finais

Este capítulo apresentou uma abordagem para geração de dados de teste utilizando algoritmos evolutivos, mais especificamente, algoritmos genéticos, e considerando critérios do teste estrutural no contexto de veículos autônomos. Um algoritmo genético adaptado para o contexto de veículos autônomos foi apresentado. Entre as adaptações para esse algoritmo estão a utilização de uma população inicial selecionada por heurística e a utilização das estratégias de combinação e mutação definidas no Capítulo 7 desta tese.

Além disso, a ferramenta PATeCa foi estendida para oferecer suporte à estratégia proposta, possibilitando que um estudo de caso fosse realizado. Esse estudo utilizou logs reais obtidos no teste de campo de um veículo autônomo e, mesmo utilizando tamanho da população e número de iterações pequenos, foi possível melhorar a cobertura obtida previamente.

A aplicação da abordagem possibilita, por meio de análise dos trechos gerados, executar o veículo autônomo *offline* e conseguir melhores coberturas que no teste de campo. Adicionalmente, seria possível pensar em cenários de teste reais para serem reproduzidos em testes de campo de forma a garantir a mesma cobertura obtida na execução *offline* e conseguir melhores coberturas que no teste de campo. Além disso, os conjuntos de dados de teste gerados poderiam ser utilizados em testes de regressão.

Conclusões

9.1 Considerações finais

Veículos autônomos representam um relevante tópico de pesquisa e vêm ganhando muita importância nos últimos anos. Essa tecnologia permite, entre outros fatores a considerar, a redução no número de acidentes de carros e a melhora no fluxo de tráfego de grandes cidades, além de ter impacto direto em atividades que dependam da mobilidade. No entanto, à medida que essa área avança, outras questões importantes surgem como, por exemplo, como garantir que esses veículos tenham sido suficientemente testados.

Neste trabalho foi proposta uma abordagem que oferece suporte ao teste estrutural de veículos autônomos. A partir do estudo do funcionamento dos veículos autônomos desenvolvidos pelo grupo de Robótica Móvel do ICMC/USP, de diversos artigos que relatem projetos semelhantes no exterior e de estudos de caso conduzidos, foram definidas estratégias para que o teste estrutural pudesse ser aplicado após a realização de testes de campo envolvendo o veículo. Dessa maneira, é possível realizar uma pós-análise do teste de campo, que é tipicamente funcional, com o objetivo de identificar trechos do código-fonte que nunca tenham sido executados para que se possa encontrar cenários de teste que possam exercitar esses trechos, aumentando a cobertura de requisitos definidos por critérios do teste estrutural. De acordo com o mapeamento sistemático conduzido, esta é uma atividade importante no processo de desenvolvimento de veículos autônomos e há pouco apoio tanto ferramental quanto conceitual.

Embora a implementação da ferramenta utilize o ambiente ROS, a estratégia pode ser facilmente adaptada para ser aplicada em outros ambientes robóticos desde que neles seja possível realizar a reprodução *offline* de um teste de campo. Além disso, como as informações de cobertura obtida do código-fonte são gravadas em arquivos XML, a adaptação da ferramenta para considerar programas escritos em linguagens diferentes de C/C++ não seria difícil. Por exemplo, para programas escritos em Python, que é uma das linguagens para as quais ROS dá suporte, já existem módulos que calculam a cobertura do código-fonte obtida, bastando apenas converter o resultado dessas coberturas para arquivos XML com o formato utilizado pela ferramenta. Outro ponto a ser considerado é que, apesar dos estudos de caso terem sido realizados com base em nuvens de pontos, a estratégia também poderia ser aplicada com outras entradas de tipos de dados simples e também com entradas que fossem estruturas de dados variantes de nuvens de pontos.

A seguir, na Seção 9.2 são apresentadas as contribuições desta tese e na Seção 9.3 são discutidos os trabalhos futuros que podem ser realizados a partir desta tese.

9.2 Contribuições desta tese

Esta tese apresenta duas contribuições principais para o teste estrutural de software no domínio de veículos autônomos: uma mais geral e conceitual e outra mais técnica e ferramental. A primeira delas é a proposta de um modelo conceitual para apoiar o teste de campo levando em consideração logs contendo as variáveis de entrada recebidas durante o teste. Algumas dessas variáveis são estruturas de dados complexas, como as nuvens de pontos. Esse modelo é base para as estratégias de redução e geração de logs bem como para as funcionalidades de cálculo e visualização da cobertura dos requisitos do teste estrutural que envolvem tanto as execuções individuais dos logs quanto as agregadas. A ferramenta PATeCA também foi desenvolvida baseada nesse modelo.

A segunda contribuição desta tese é a proposta de um algoritmo genético para gerar dados de teste no contexto de veículos autônomos que tem como variáveis de entrada nuvens de pontos, que são provenientes de imagens do ambiente captadas por diversos tipos de câmeras. Para que se tornasse factível usar esse algoritmo com bons resultados foram definidas heurísticas para seleção de um conjunto menor de variáveis de entradas contidas nos logs a fim de ser utilizado como população inicial. Além disso, também foram definidas estratégias de combinação e de mutação para criar nuvens de pontos modificadas e com chances de aumentarem as coberturas obtidas originalmente com o teste de campo. Esse algoritmo foi também implementado na ferramenta PATeCa e estudos de caso com um software controlador de um veículo real foram realizados para validar o algoritmo, as heurísticas e as estratégias definidas na tese.

9.3 Trabalhos futuros

Há muitas pesquisas que são motivadas por esta tese e que podem ser realizadas no futuro pela autora e por outros pesquisadores do grupo de pesquisa em engenharia de software do ICMC.

Primeiramente, seria muito importante realizar mais estudos de caso com outros programas controladores de veículos autônomos com o objetivo de comprovar se os mesmos resultados, ou resultados semelhantes, são obtidos. Também poderiam ser considerados outros tipos de veículos autônomos, como os aquáticos e aéreos. Além disso, tanto para as heurísticas definidas no Capítulo 6 quanto para o algoritmo genético definido no Capítulo 8, também seriam interessantes mais estudos de caso que envolvessem a utilização de um conjunto de logs particionados em trechos de outros tamanhos. Para o algoritmo genético, também seria importante mais estudos de caso envolvendo uma população e um número de iterações maiores. Também poderiam ser realizados experimentos para analisar a eficiência em detectar falhas da abordagem e identificar quais os tipos de defeitos é mais propenso a detectar. Para isso, poderiam ser injetados artificialmente um número de defeitos utilizando mutantes, por exemplo.

Embora as nuvens de pontos sejam uma das estruturas de dados mais complexas que um software controlador nesse domínio possa receber, um outro trabalho futuro seria a adaptação do algoritmo de geração para receber outros tipos de variáveis de entrada bem como considerar mais de uma variável de entrada. Por exemplo, para alguns dos estudos de caso realizados só não foi possível aumentar ainda mais a cobertura porque as instruções não exercitadas dependiam da variável correspondente às coordenadas do GPS. Sendo assim, seria interessante que o algoritmo considerasse, além das nuvens de pontos, a mutação e combinação das coordenadas recebidas pelo GPS.

Outro trabalho futuro seria definir novas estratégias de combinação e de mutação das nuvens de pontos. Outros tipos de algoritmos baseados em busca poderiam ser estudados e implementados, permitindo comparar com os resultados do algoritmo genético proposto nesta tese. Outros critérios do teste estrutural poderiam ser considerados no algoritmo como, por exemplo, os critérios de fluxo de dados.

Poderiam ser pesquisadas estratégias que envolvessem a conversão de um log de teste gerado artificialmente e que aumentou a cobertura em um cenário de teste real. Além disso, como a abordagem pode gerar vários dados de teste que não poderiam ser reproduzidos na vida real, seria interessante definir operadores de mutação e combinação que garantissem que os dados de teste gerados tivessem um equivalente físico ou criar estratégias que fizessem a seleção dos logs gerados que possuem correspondência física.

Seria também interessante aplicar as mesmas técnicas e algoritmos e adaptar a ferramenta PAtCa para outros tipos de sistemas embarcados que têm a mesma característica de controle dos veículos autônomos: uma iteração mais externa em que dados do ambiente são lidos para que o sistema possa tomar uma decisão. São exemplos de aplicações semelhantes às do domínio de dispositivos de saúde, como marca-passos, e dispositivos de segurança doméstica.

Outra linha de pesquisa seria a de oráculos de teste, ou seja, como definir funções de qualidade para decidir se a implementação é compatível com a especificação de alto nível do sistema. Além disso, poderiam ser conduzidas pesquisas que aplicassem as mesmas estratégias aqui propostas na fase de simulação. Ou seja, definir operadores que manipulassem o mundo virtual criado pelos simuladores, por exemplo, adicionando obstáculos ou alterando as condições de luz, ao invés de manipular as nuvens de pontos previamente gravadas.

Referências

- Bacha, A.; Bauman, C.; Faruque, R.; Fleming, M.; Terwelp, C.; Reinholtz, C.; Hong, D.; Wicks, A.; Alberi, T.; Anderson, D.; Cacciola, S.; Currier, P.; Dalton, A.; Farmer, J.; Hurdus, J.; Kimmel, S.; King, P.; Taylor, A.; Covern, D. V.; Webster, M. Odin: Team victortango's entry in the darpa urban challenge. *Journal of Field Robotics*, v. 25, n. 8, p. 467–92, 2008a.
- Bacha, A.; Bauman, C.; Faruque, R.; Fleming, M.; Terwelp, C.; Reinholtz, C.; Hong, D.; Wicks, A.; Alberi, T.; Anderson, D.; Cacciola, S.; Currier, P.; Dalton, A.; Farmer, J.; Hurdus, J.; Kimmel, S.; King, P.; Taylor, A.; Covern, D. V.; Webster, M. Odin: Team victortango's entry in the darpa urban challenge. *Journal of Field Robotics*, v. 25, n. 8, p. 467–492, 2008b.
- Barbosa, E. F.; Chaim, M. L.; Vincenzi, A. M. R.; Delamaro, M. E.; Jino, M.; Maldonado, J. C. *Teste estrutural*. Campus Elsevier, p. 47–76, 2007.
- Berger, C. Improving scenario selection for simulations by run-time control-flow analysis. In: *Proceedings of the 2013 Summer Computer Simulation Conference, SCSC '13*, Vista, CA: Society for Modeling & Simulation International, 2013, p. 25:1–25:8 (*SCSC '13*,).
- Braione, P.; Denaro, G.; Mattavelli, A.; Vivanti, M.; Muhammad, A. Software testing with code-based test generators: Data and lessons learned from a case study with an industrial software component. *Software Quality Control*, v. 22, n. 2, p. 311–333, 2014.
- Burke, E. K.; Kendall, G. *Search methodologies: Introductory tutorials in optimization and decision support techniques*. 2nd ed. Springer Publishing Company, Incorporated, 2013.

- Costa, J. C.; Monteiro, J. C. Coverage-directed observability-based validation for embedded software. *Trans. on Design Automation of Electronic Systems*, v. 18, n. 2, p. 1–20, 2013.
- Crane, Carl D., I.; Armstrong, David G., I.; Touchton, R.; Galluzzo, T.; Solanki, S.; Lee, J.; Kent, D.; Ahmed, M.; Montane, R.; Ridgeway, S.; Velat, S.; Garcia, G.; Griffis, M.; Gray, S.; Washburn, J.; Routson, G. Team cimar’s navigator: An unmanned ground vehicle for the 2005 darpa grand challenge. In: Buehler, M.; Iagnemma, K.; Singh, S., eds. *The 2005 DARPA Grand Challenge*, v. 36 de *Springer Tracts in Advanced Robotics*, Springer Berlin Heidelberg, p. 311–347, 2007.
- Delamaro, M. E.; Barbosa, E. F.; Vincenzi, A. M. R.; ; Maldonado, J. C. *Teste de mutação*. Campus Elsevier, p. 77–118, 2007.
- Didrich, K.; Herbst, S.; Vieira, M. Applying model-based testing to a train control system. *INFORMATIK 2006 - Informatik fur Menschen, Beitrage der 36. Jahrestagung der Gesellschaft fur Informatik e.V. (GI)*, v. 2, p. 249–256, 2006.
- Doganay, K.; Bohlin, M.; Sellin, O. Search based testing of embedded systems implemented in iec 61131-3: An industrial case study. In: *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, 2013, p. 425–432.
- Doganay, K.; Eldh, S.; Afzal, W.; Bohlin, M. Search-based testing for embedded telecom software with complex input structures. In: Merayo, M.; de Oca, E., eds. *Testing Software and Systems*, v. 8763 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, p. 205–210, 2014.
- Drusinsky, D.; Shing, M.-T.; Demir, K. A. Creating and validating embedded assertion statecharts. *IEEE Distributed Systems Online*, v. 8, n. 5, 2007.
- Ebert, C.; Salecker, J. Embedded software technologies and trends. *IEEE Software*, v. 26, n. 3, p. 14–18, 2009.
- Fernandes, L. C.; Souza, J. R.; Pessin, G.; Shinzato, P. Y.; Sales, D. O.; Mendes, C. C. T.; Prado, M.; Klaser, R. L.; Magalhães, A. C.; Hata, A. Y.; Pigatto, D. F.; Branco, K. R. L. J. C.; Jr., V. G.; Osório, F. S.; Wolf, D. F. Carina intelligent robotic car: Architectural design and applications. *Journal of Systems Architecture - Embedded Systems Design*, v. 60, n. 4, p. 372–392, 2014.
- Disponvel em <http://dx.doi.org/10.1016/j.sysarc.2013.12.003>

- Fernandes, L. C.; Souza, J. R.; Shinzato, P. Y.; Pessin, G.; Mendes, C. C. T.; Osório, F. S.; Wolf, D. F. Intelligent robotic car for autonomous navigation: Platform and system architecture. In: *CBSEC 2012 - II Brazilian Conference on Critical Embedded Systems*, 2012, p. 12 – 7.
- Freescale Codewarrior development tools. Online, disponível em http://www.freescale.com/webapp/sps/site/homepage.jsp?code=CW_HOME&tid=vancodewarrior - Último acesso em 26/02/2015, 2015.
- Gordon, F.; Arcuri, A. The seed is strong: Seeding strategies in search-based software testing. In: *Proc. of the 5th IEEE Int. Conf. on Software Testing, Verification and Validation*, Montreal, CA: IEEE Computer Society, 2012, p. 121–30.
- Graf, P.; Muller-Glaser, K. D.; Reichmann, C. Nonintrusive black- and white-box testing of embedded systems software against uml models. In: *Proc. of the 18th IEEE/IFIP Int. Workshop on Rapid System Prototyping*, Washington, DC, USA: IEEE Computer Society, 2007, p. 130–138.
- Gross, H.; Kruse, P. M.; Wegener, J.; Vos, T. Evolutionary white-box software test with the evotest framework: A progress report. In: *ICSTW '09: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, Washington, DC, USA: IEEE Computer Society, 2009, p. 111–120.
- Guan, J.; Offutt, J.; Ammann, P. An industrial case study of structural testing applied to safety-critical embedded software. In: *Proc. of the ACM/IEEE int. Symposium on Empirical software engineering*, New York, NY, USA: ACM, 2006, p. 272–77.
- Harman, M.; Hassoun, Y.; Lakhotia, K.; McMinn, P.; Wegener, J. The impact of input domain reduction on search-based test data generation. In: *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, New York, NY, USA: ACM, 2007, p. 155–164 (*ESEC-FSE '07*,).
- Harrold, M. J.; Gupta, R.; Soffa, M. L. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering Methodology*, v. 2, n. 3, p. 270–285, 1993.
- He, N.; Rummer, P.; Kroening, D. Test-case generation for embedded simulink via formal concept analysis. In: *Design Automation Conference (DAC), 2011 48th ACM/E-DAC/IEEE*, 2011, p. 224–229.

- Hla, K.; Choi, Y.; Park, J. S. Applying particle swarm optimization to prioritizing test cases for embedded real time software retesting. In: *Proc. of the IEEE 8th Int. Conference on Computer and Information Technology Workshops*, 2008, p. 527–532.
- IEEE *Standard glossary of software engineering terminology*. Standard 610.12, Institute of Electric and Electronic Engineers, 1990.
- Jo, K.; Kim, J.; Kim, D.; Jang, C.; Sunwoo, M. Development of autonomous car - part i: Distributed system architecture and development process. *Industrial Electronics, IEEE Transactions on*, v. 61, n. 12, p. 7131–7140, 2014.
- Jung, C.; Osório, F. S.; Kelber, C.; Heinen, F. Computação embarcada: Projeto e implementação de veículos autônomos inteligentes. *Anais do CSBC-05 XXIV Jornada de Atualização em Informática*, v. 1, p. 1358 – 406, 2005.
- Jürjens, J.; Reiß, D.; Trachtenherz, D. Model-based quality assurance of automotive software. In: *Proc. of the 11th Int. Conf. on Model Driven Engineering Languages and Systems*, Berlin, Heidelberg: Springer-Verlag, 2008, p. 858–73.
- Kirner, R. SCCP/x: a compilation profile to support testing and verification of optimized code. In: *Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, New York, NY, USA: ACM, 2007, p. 38–42.
- Kitchenham, B.; Charters, S. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Relatório Técnico EBSE 2007-001, Keele University and Durham University Joint Report, 2007.
- Koong, C.-S.; Lai, H.-J.; Chang, C.-H.; Chu, W. C.; Hsueh, N.-L.; Hsiung, P.-A.; Shih, C.; Yang, C.-T. Supporting tool for embedded software testing. In: *Proc. of the 10th Int. Conference on Quality Software, QSIC '10*, Washington, DC, USA: IEEE Computer Society, 2010, p. 481–87 (*QSIC '10*,).
- Koong, C.-S.; Shih, C.; Hsiung, P.-A.; Lai, H.-J.; Chang, C.-H.; Chu, W. C.; Hsueh, N.-L.; Yang, C.-T. Automatic testing environment for multi-core embedded software-ates. *J. Syst. Softw.*, v. 85, n. 1, p. 43–60, 2012.
- Kukulj, S.; Marinkovic, V.; Popovic, M.; Bognar, S. Selection and prioritization of test cases by combining white-box and black-box testing methods. In: *Engineering of Computer Based Systems (ECBS-EERC), 2013 3rd Eastern European Regional Conference on the*, 2013, p. 153–156.
- Larman, C. *Utilizando uml e padrões*. Bookman, 695 p., 2007.

- Lemos, O. A. L.; Franchin, I. G.; Masiero, P. C. Integration testing of object-oriented and aspect-oriented programs: A structural pairwise approach for java. *Science of Computer Programming*, v. 74, n. 10, p. 861–78, 2009.
- Lemos, O. A. L.; Vincenzi, A. M. R.; Maldonado, J. C.; Masiero, P. C. Control and data flow structural testing criteria for aspect-oriented programs. *Journal of Systems and Software*, v. 80, n. 6, p. 862–82, 2007.
- Li, J. J.; Weiss, D. M.; Yee, H. An automatically-generated run-time instrumenter to reduce coverage testing overhead. In: *Proc. of the 3rd Int. Workshop on Automation of Software Test*, New York, NY, USA: ACM, 2008, p. 49–56.
- Luettel, T.; Himmelsbach, M.; Wuensche, H.-J. Autonomous ground vehicles - concepts and a path to the future. *Proceedings of the IEEE*, v. 100, n. Special Centennial Issue, p. 1831–1839, 2012.
- Maldonado, J. C. *Crterios potenciais usos: Uma contribuio ao teste estrutural de software*. Tese de Doutorado, DCA-FEE-UNICAMP, Campinas, SP - Brasil, 1991.
- McMinn, P. Search-based software test data generation: a survey. *Software Testing Verification and Reliability*, v. 14, p. 105–56, 2004.
- McMinn, P.; Shahbaz, M.; Stevenson, M. In: *Search-Based Test Input Generation for String Data Types Using the Results of Web Queries*, N: IEEE, 2012, p. 141–50.
- Mendes, C. C. T. *Navigation of mobile robots using stereo vision*. Dissertao de Mestrado, ICMC-USP, So Carlos, SP, BR, 2012.
Disponvel em <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-18062012-162436/en.php>
- Mendes, C. C. T.; Wolf, D. F. In: *Real Time Autonomous Navigation and Obstacle Avoidance Using a Semi-global Stereo Method*, New York, USA: ACM, 2013, p. 235–36.
- Micskei, Z.; Szatmri, Z.; Olh, J.; Majzik, I. *A concept for testing robustness and safety of the context-aware behaviour of autonomous systems*, v. 7327 de *Lecture Notes in Computer Science* Springer Berlin / Heidelberg, p. 504–513, 2012.
- Morton, S. Enhanced software design reusability in safety-critical embedded applications using automated structural test generation. In: *IEEE Systems Readiness Technology Conference*, IEEE, 1999, p. 755 – 67.
- Myers, G. *The art of software testing*. John Wiley and Sons, 2004.

- Neves, V. N.; Lemos, O. A.; Masiero, P. C. Teste estrutural de integração nível 1 de programas orientados a objetos e aspectos. In: *III Latin American Workshop on Aspect-Oriented Software Development*, Fortaleza, CE, 2009, p. 31–38.
- Nilsson, R.; Henriksson, D. Test case generation for flexible real-time control systems. In: *10th IEEE Conference on Emerging Technologies and Factory Automation*, IEEE, 2005, p. 8 pp. –730.
- Nothdurft, T.; Hecker, P.; Ohl, S.; Saust, F.; Maurer, M.; Reschka, A.; Bohmer, J. Stadtpilot: First fully autonomous test drives in urban traffic. In: *Intelligent Transportation Systems (ITSC), 2011 14th International IEEE Conference on*, 2011, p. 919–924.
- Petersen, K.; Feldt, R.; Mujtaba, S.; Mattsson, M. Systematic mapping studies in software engineering. In: *Proc. of the 12th Int. Conf. on Evaluation and Assessment in Software Engineering*, Swinton, UK: British Computer Society, 2008, p. 68–77.
- Prefuse The prefuse visualization toolkit. Online, available: <http://prefuse.org/> - last access in em 26/02/2015, 2014.
- Rapps, S.; Weyuker, E. J. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, v. SE-11, n. 4, p. 367–375, 1985.
- Rauskolb, F. W.; Berger, K.; Lipski, C.; Magnor, M.; Cornelsen, K.; Effertz, J.; Form, T.; Graefe, F.; Ohl, S.; Schumacher, W.; Wille, J.-M.; Hecker, P.; Nothdurft, T.; Doering, M.; Homeier, K.; Morgenroth, J.; Wolf, L.; Basarke, C.; Berger, C.; Gülke, T.; Klose, F.; Rumpe, B. Caroline: An autonomously driving vehicle for urban environments. *Journal of Field Robotics*, v. 25, n. 9, p. 674–724, 2008.
- Rocha, A. R. C.; Maldonado, J. C.; Weber, K. C. *Qualidade de software: Teoria e prática*. São Paulo, SP: Prentice Hall, 2001.
- Romero, R. A. F.; Prestes, E.; Osório, F.; Wolf, D. *Robótica móvel*. Rio de Janeiro: Editora LTC, 302 p., 2014.
- ROS.org Documentation - ros. Online, disponível em <http://www.ros.org> - último acesso em 26/02/2015, 2015.
- RTCA.org RTCA - The Gold Standard for Aviation Since 1935. Online, disponível em <http://www.rtca.org> - Último acesso em 26/02/2015, 2015.
- Shin, D.; Jee, E.; Bae, D.-H. Empirical evaluation on fbd model-based test coverage criteria using mutation analysis. In: *Proceedings of the 15th International Conference*

- on Model Driven Engineering Languages and Systems*, MODELS'12, Berlin, Heidelberg: Springer-Verlag, 2012, p. 465–479 (*MODELS'12*,).
- Siegiwart, R.; Nourbakhsh, I. R. *Introduction to autonomous mobile robots*. Scituate, MA, USA: Bradford Company, 2004.
- Suganuma, N.; Uozumi, T. Development of an autonomous vehicle: System overview of test ride vehicle in the Tokyo Motor Show 2011. In: *Proc. of the Int. Conf. on Instrumentation, Control, Information Technology and System Integration*, Japan: Akita University, 2012, p. 215–18.
- Tai, K.-C. Predicate-based test generation for computer programs. In: *Proc. of the 15th int. Conference on Software Engineering*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1993, p. 267–76.
- Team, G. Gcov - using the gnu compiler collection (gcc). Online, disponível em <http://gcc.gnu.org/onlinedocs/gcc/Gcov.html> - Último acesso em 26/02/2015, 2014.
- Terry, N. trucov - the true c and c++ test coverage analysis tool. Online, disponível em <http://code.google.com/p/trucov/> - Último acesso em 26/02/2015, 2014.
- Thrun, S.; Montemerlo, M.; Dahlkamp, H.; Stavens, D.; Aron, A.; Diebel, J.; Fong, P.; Gale, J.; Halpenny, M.; Hoffmann, G.; Lau, K.; Oakley, C.; Palatucci, M.; Pratt, V.; Stang, P.; Strohband, S.; Dupont, C.; Jendrossek, L.-E.; Koelen, C.; Markey, C.; Rummel, C.; van Niekerk, J.; Jensen, E.; Alessandrini, P.; Bradski, G.; Davies, B.; Ettinger, S.; Kaehler, A.; Nefian, A.; Mahoney, P. Stanley: The robot that won the darpa grand challenge. *Journal of Field Robotics*, v. 23, p. 661–92, 2006.
- Tian, P.; Wang, J.; Leng, H.; Qiang, K. Construction of distributed embedded software testing environment. *Int. Conf. on Intelligent Human-Machine Systems and Cybernetics*, v. 1, n. 8, p. 470 – 73, 2009.
- Ting, D.; Zhongmin, W. Research on instrumentation technology for the embedded assembly software testing based on host computer. In: *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, 2011, p. 1906–1909.
- Tlili, M.; Wappler, S.; Sthamer, H. Improving evolutionary real-time testing. In: *Proc. of the 8th Annual Conf. on Genetic and Evolutionary Computation*, New York, NY, USA, 2006, p. 1917–24.
- Tracey, N.; Clark, J.; McDermid, J.; Mander, K. Systems engineering for business process change. cap. A Search-based Automated Test-data Generation Framework

- for Safety-critical Systems, New York, NY, USA: Springer-Verlag New York, Inc., p. 174–213, 2002.
- Trepagnier, P. G.; Nagel, J.; Kinney, P. M.; Koutsougeras, C.; Dooner, M. Kat-5: Robust systems for autonomous vehicle navigation in challenging and unknown terrain. *Journal of Field Robotics*, v. 23, n. 8, p. 509–26, 2006.
- Urmson, C.; Anhalt, J.; Bagnell, D.; ; Baker, C.; Bittner, R.; Clark, M. N.; Dolan, J.; Duggins, D.; Galatali, T.; Geyer, C.; Gittleman, M.; Harbaugh, S.; Hebert, M.; Howard, T. M.; Kolski, S.; Kelly, A.; Likhachev, M.; McNaughton, M.; Miller, N.; Peterson, K.; Pilnick, B.; Rajkumar, R.; Rybski, P.; Salesky, B.; Seo, Y.-W.; Singh, S.; Snider, J.; Stentz, A.; Whittaker, W. R.; Wolkowicki, Z.; Ziglar, J.; Bae, H.; Brown, T.; Demitrish, D.; Litkouhi, B.; Nickolaou, J.; Sadekar, V.; Zhang, W.; Struble, J.; Taylor, M.; Darms, M.; Ferguson, D. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, v. 25, p. 425–66, 2008.
- Urmson, C.; Ragusa, C.; Ray, D.; Anhalt, J.; Bartz, D.; Galatali, T.; Gutierrez, A.; Johnston, J.; Harbaugh, S.; “Yu” Kato, H.; Messner, W.; Miller, N.; Peterson, K.; Smith, B.; Snider, J.; Spiker, S.; Ziglar, J.; “Red” Whittaker, W.; Clark, M.; Koon, P.; Mosher, A.; Struble, J. A robust approach to high-speed navigation for unrehearsed desert terrain. *Journal of Field Robotics*, v. 23, n. 8, p. 467–508, 2006.
- Vergilio, S. R.; Maldonado, J. C.; Jino, M. *Geração de dados de teste*. Campus Elsevier, p. 269–292, 2007.
- Vincenzi, A. M. R.; Delamaro, M. E.; Maldonado, J. C.; Wong, W. E. Establishing structural testing criteria for java bytecode. *Software Practice and Experience*, v. 36, n. 14, p. 1513–41, 2006.
- Wegener, J.; Buhr, K.; Pohlheim, H. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In: *Proc. of the Genetic and Evolutionary Computation Conference*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, p. 1233–40.
- Wegener, J.; Sthamer, H.; Jones, B. F.; Eyres, D. E. Testing real-time systems using genetic algorithms. *Software Quality Control*, v. 6, p. 127–35, 1997.
- White, L. J.; Cohen, E. I. A domain strategy for computer program testing. *IEEE Trans. on Software Engineering*, v. 6, p. 247–57, 1980.

- Windisch, A. Search-based test data generation from stateflow statecharts. In: *Proc. of the 12th Annual Conf. on Genetic and Evolutionary Computation*, GECCO '10, New York, NY, USA: ACM, 2010, p. 1349–56 (*GECCO '10*,).
- Wolf, D. F.; Simões, E. V.; Osório, F. S.; Junior, O. T. Robótica móvel inteligente: da simulação às aplicações no mundo real. In: *Anais do CSBC-09 XXIX Jornada de Atualização em Informática*, 2009.
- Wu, X.; Li, J. J.; Weiss, D.; Lee, Y. Coverage-based testing of embedded systems. In: *Proc. of the 2nd Int. Workshop on Automation of Software Test*, Washington, DC, USA: IEEE Computer Society, 2007, p. 20 – 26.
- Yang, Q.; Li, J. J.; Weiss, D. A survey of coverage based testing tools. In: *Proc. of the Int. Workshop on Automation of Software Test*, New York, NY, USA: ACM, 2006, p. 99–103.
- Yu, Y.; Jones, J. A.; Harrold, M. J. An empirical study of the effects of test-suite reduction on fault localization. In: *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, New York, NY, USA: ACM, 2008, p. 201–210 (*ICSE '08*,).
- Zhi, X.; Zhang, J. Critical path coverage testing of embedded software based on lcsaj. In: *Advanced Research and Technology in Industry Applications (WARTIA), 2014 IEEE Workshop on*, 2014, p. 1163–1166.
- Zhu, H.; Hall, P.; May, J. Software unit test coverage and adequacy. *ACM Computing Surveys*, v. 29, n. 4, p. 366–427, 1997.