

Paulo Marcos Siqueira Bueno

Geração de Dados de Teste
Orientada à Diversidade com
o uso de Meta-Heurísticas

CAMPINAS
2012

Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

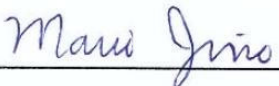
Paulo Marcos Siqueira Bueno

Geração de Dados de Teste Orientada à
Diversidade com o uso de Meta-Heurísticas

Orientador: Prof. Dr. Mario Jino

Tese apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas, para obtenção do Título de Doutor em Engenharia Elétrica. Área de concentração: Engenharia de Computação.

Este exemplar corresponde à versão final da tese defendida pelo aluno Paulo Marcos Siqueira Bueno e orientada pelo Prof. Dr. Mario Jino.


Prof. Mario Jino

Campinas, 2012

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE -
UNICAMP

B862g Bueno, Paulo Marcos Siqueira
 Geração de dados de teste orientada à diversidade
 com o uso de meta-heurísticas / Paulo Marcos Siqueira
 Bueno. --Campinas, SP: [s.n.], 2012.

 Orientador: Mario Jino.
 Tese de Doutorado - Universidade Estadual de
Campinas, Faculdade de Engenharia Elétrica e de
Computação.

 1. Algoritmos genéticos. 2. Engenharia de software.
3. Software - Validação. I. Jino, Mario, 1943-. II.
Universidade Estadual de Campinas. Faculdade de
Engenharia Elétrica e de Computação. III. Título.

Título em Inglês: Diversity oriented test data generation using meta-
heuristic techniques

Palavras-chave em Inglês: Genetic algorithms, Software engineering,
Software - Validation

Área de concentração: Engenharia de Computação

Titulação: Doutor em Engenharia Elétrica

Banca examinadora: Marcos Lordello Chaim, Silvia Regina Vergilio,
Eliane Martins, Fernando José Von Zuben

Data da defesa: 21-09-2012

Programa de Pós Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE DOUTORADO

Candidato: Paulo Marcos Siqueira Bueno

Data da Defesa: 21 de setembro de 2012

Título da Tese: "Geração de Dados de Teste Orientada à Diversidade com o Uso de Meta-Heurísticas"

Prof. Dr. Mario Jino (Presidente): Mario Jino

Prof. Dr. Marcos Lordello Chaim: Marcos Lordello Chaim

Profa. Dra. Silvia Regina Vergilio: Silvia Regina Vergilio

Profa. Dra. Eliane Martins: Eliane Martins

Prof. Dr. Fernando José Von Zuben: Fernando José Von Zuben

Dedico este trabalho ao Prof. Mario Jino

Agradecimentos

Aos meus pais, pelo privilégio de ter tido um ambiente favorável ao meu desenvolvimento como pessoa e à formação da minha base profissional. Aos meus irmãos, Débora e Samuel, pelo estímulo e ajuda. De forma especial a minha mãe, Edna, pelo apoio incondicional e pelo carinho.

À Júnia, pelo apoio no dia a dia, pelo amor e por fazer com que conquistas tenham mais sentido. E também à Valentina e à Giordana.

Ao Prof. Mario Jino, pela amizade, apoio e orientação fundamentais para a realização deste trabalho.

Aos colegas do Centro de Tecnologia da Informação Renato Archer, pelo incentivo e por propiciarem condições ao meu desenvolvimento profissional. Em especial, ao Adalberto Crespo e ao Miguel Argollo, por compartilharem suas experiências e pela oportunidade do meu engajamento em projetos no CTI.

Ao Prof. W. Eric Wong, por ter me recebido em programa “sanduiche” na Universidade do Texas em Dallas e na Telcordia Technologies, New Jersey.

Aos membros da banca examinadora, professores (as): Marcos Chaim, Silvia Vergilio, Eliane Martins e Fernando Von Zuben pelas importantes críticas e sugestões.

Ao CNPq e à CAPES (BEX 1544/01-2), pelo apoio financeiro.

*“O saber se aprende com os mestres.
A sabedoria, só com o corriqueiro da
vida.”*

Cora Coralina

*“A alegria não chega apenas no encontro
do achado, mas faz parte do processo da
busca. E ensinar e aprender não pode
dar-se fora da procura, fora da boniteza
e da alegria.”*

Paulo Freire.

*“It is probably true quite generally that in the history of
human thinking the most fruitful developments
frequently take place at those points where two different
lines of thought meet. These lines may have their roots
in quite different parts of human nature, in different
times or different cultural environments or different
religious traditions: hence if they actually meet, that is, if
they are at least so much related to each other that a
real interaction can take place, then one may hope that
new and interesting developments may follow.”*

Werner Heisenberg

*“The most exciting phrase to
hear in science, the one that
heralds new discoveries, is not
'Eureka!' but 'That's funny...'. ”*

Isaac Asimov

Resumo

Técnicas e critérios de teste de software estabelecem elementos requeridos a serem exercitados no teste. A geração de dados de teste visa selecionar dados de teste, do domínio multidimensional de entrada do software, para satisfazer um critério. Uma linha de trabalhos para a geração de dados de teste utiliza meta-heurísticas para buscar, no espaço de possíveis entradas do software, aquelas que satisfaçam um determinado critério, área referida como Teste de Software Baseado em Buscas. Esta tese propõe uma nova técnica, a Geração de Dados de Teste Orientada à Diversidade (*Diversity Oriented Test Data Generation – DOTG*). Esta técnica incorpora a intuição, encontrada em bons projetistas de teste, de que a variedade, ou diversidade, dos dados de teste tem um papel relevante para a completeza, ou qualidade, do teste realizado. São propostas diferentes perspectivas para a diversidade do teste; cada perspectiva leva em consideração um tipo de informação distinto para avaliar a diversidade. É definido também um meta-modelo para guiar o desenvolvimento das perspectivas da DOTG. É desenvolvida a perspectiva do domínio de entrada do software para a diversidade (DOTG-ID), que considera a posição dos dados de teste neste domínio para calcular a diversidade do conjunto de teste. São propostas uma medida de distância entre dados de teste e uma medida de diversidade de conjuntos de teste. São desenvolvidas três meta-heurísticas para a geração automática de dados de alta diversidade: a SA-DOTG, baseada em Recozimento Simulado; a GA-DOTG, baseada em Algoritmos Genéticos; e a SR-DOTG, baseada na dinâmica de sistemas de partículas eletricamente carregadas. A avaliação empírica da DOTG-ID inclui: uma simulação Monte Carlo, realizada com o objetivo de estudar a influência de fatores na eficácia da técnica; e um experimento com programas, realizado para avaliar o efeito da diversidade dos conjuntos de teste na cobertura alcançada, medida com respeito a critérios de teste baseados em análise de fluxos de dados e no critério baseado em defeitos Análise de Mutantes. Os resultados das avaliações, significativos estatisticamente, indicam que na maioria das situações os conjuntos de alta diversidade atingem eficácia e valores de cobertura maiores do que os alcançados pelos conjuntos gerados aleatoriamente, de mesmo tamanho.

Palavras chave: teste de software, geração de dados de teste, diversidade, Recozimento Simulado, Algoritmos Genéticos, Repulsão Simulada, avaliação empírica.

Abstract

Software testing techniques and criteria establish required elements to be exercised during testing. Test data generation aims at selecting test data from the multidimensional software's input domain to satisfy a given criterion. A set of works on test data generation apply metaheuristics to search in the space of possible inputs for the software for those inputs that satisfy a given criterion. This field is named Search Based Software Testing. This thesis proposes a new technique, the Diversity Oriented Test Data Generation – DOTG. This technique embodies the intuition, which can be found in good testers, that the variety, or diversity, of test data used to test a software has some relation with the completeness, or quality, of the testing performed. We propose different perspectives for the test diversity concept; each one takes into account a different kind of information to evaluate the diversity. A metamodel is also defined to guide the development of the DOTG perspectives. We developed the Input Domain perspective for diversity (DOTG-ID), which considers the positions of the test data in the software input domain to compute a diversity value for the test sets. We propose a measure of distance between test data and a measure of diversity of test sets. For the automatic generation of high diversity test sets three metaheuristics were developed: the SA-DOTG based on Simulated Annealing; the GA-DOTG based on Genetic Algorithms, and the SR-DOTG, based on the dynamics of particle systems electrically charged. The empirical evaluation of DOTG-ID includes: a Monte Carlo simulation performed to study the influence of factors on the technique's effectiveness, and an experiment with programs, carried out to evaluate the effect of the test sets diversity on the attained coverage values, measured with respect to data-flow coverage and to mutation coverage. The evaluation results are statistically significant, pointing out that in most of cases the test sets with high diversity reach effectiveness and coverage values higher than the ones reached by randomly generated test sets of the same size.

Keywords: software testing, test data generation, diversity, Simulated Annealing, Genetic Algorithms, Simulated Repulsion, empirical evaluation.

Lista de Figuras

Figura 1. Cadeia de Eventos Para a Falha do Software.....	14
Figura 2. Classes de Equivalência e Dados de Teste	30
Figura 3. Valores Limite e Dados de Teste.....	32
Figura 4. Código Fonte e Respectivo Grafo de Fluxo de Controle.....	39
Figura 5. Comando if-then-else e Respectivo Grafo de Fluxo de Controle	42
Figura 6. Comando if-then e Respectivo Grafo de Fluxo de Controle.....	42
Figura 7. Comandos if-then em Sequência e Respectivo Grafo de Fluxo de Controle.....	43
Figura 8. exemplo de aplicação da técnica baseada em análise de fluxo de dados.....	46
Figura 9. Projeções bidimensionais de uma região de falha dependente de um conjunto de variáveis de entrada	56
Figura 10. Projeções bidimensionais de uma região de falha dependente da relação entre variáveis de entrada	57
Figura 11. “Cristal de Erro” identificado em software da NASA	58
Figura 12. Padrões de Regiões de falha dos tipos: ponto, faixa e bloco.	59
Figura 13. Trechos de programas que geram falhas dos tipos: ponto (a), faixa (b) e bloco(c)	60
Figura 14. Geração de dados de teste por meio de execução simbólica.....	68
Figura 15. Geração de dados de teste usando a abordagem dinâmica.....	70
Figura 16. Recozimento Simulado (versão para maximização).....	78
Figura 17. Algoritmo Genético	79
Figura 18. Elementos de uma solução de geração de dados de teste baseada em buscas	87
Figura 19. Procedimento genérico para a geração de dados de teste baseada em buscas	109
Figura 20. Exemplo de cálculo da diversidade.....	135
Figura 21. Alocação de dados de teste e valores de diversidade do conjunto de teste.....	136
Figura 22. Perda de informação de diversidade	139
Figura 23. Conjunto RTS (esquerda) e conjunto DOTS-ID (direita).....	139
Figura 24. Regiões de falha e conjuntos RTS (a) e DOTS (b).....	145
Figura 25. SA-DOTG.....	149
Figura 26. GA-DOTG	152
Figura 27. Meta-heurística Repulsão Simulada	154
Figura 28. Cálculo de forças eletrostáticas entre dados de teste	156
Figura 29. Movimentos de dados de teste	157
Figura 30. Detalhamento da meta-heurística Repulsão Simulada.....	159
Figura 31. Eficácia de conjuntos DOTS e RTS.....	183
Figura 32. Razão de eficácia entre conjuntos DOTS e RTS	185
Figura 33. Distância média entre dados de teste do conjunto DOTS e eficácia.....	187
Figura 34. Razão de eficácia e tamanho de região de falha	189
Figura 35. Contexto do procedimento de avaliação	192
Figura 36. Programa exemplo, GFC e GDC	263
Figura 37. Re-expressão de dados de entrada na diversidade de dados	283
Figura 38. Ilustração do modelo espaço de formas, Perelson e Oster (1979)	307
Figura 39. Exemplo de diversidade de estados internos de execução.....	318

Lista de Tabelas

Tabela 1. Valores de diversidade e de somatório de distâncias	137
Tabela 2. Número de iterações e valores de diversidade	176
Tabela 3. expint - resultados de cobertura para DOTS e RTS	201
Tabela 4. Resultados estatísticos (valores p) para expint	201
Tabela 5. tritype resultados de cobertura para DOTS e RTS	202
Tabela 6. Resultados estatísticos (valores p) para tritype	202
Tabela 7. tcas-main resultados de cobertura para DOTS e RTS	204
Tabela 8. Resultados estatísticos (valores p) para tcas-main	205
Tabela 9. Resultados de cobertura para alt-sep-test e escore de mutação para main e alt-sep-test.	205
Tabela 10. Resultados estatísticos (valores p) para alt-sep-test, main	205
Tabela 11. Razão entre a cobertura de conjuntos DOTS e conjuntos RTS.....	207
Tabela 12. Função objetivo de Michel et al (2001).....	261
Tabela 13. Analogia entre o reconhecimento imune e a revelação de defeitos.....	313

Publicações Relacionadas

Bueno, P.M.S., Jino, M. Wong, W.E., “Diversity Oriented Test Data Generation Using Metaheuristic Search Techniques,” Information Sciences, Elsevier, (<http://dx.doi.org/10.1016/j.ins.2011.01.025>), 2011.

Vergilio, S.R., Bueno, P.M.S., Dias Neto, A. C., Martins E., Vincenzi A., “Geração de Dados de Teste: Principais Conceitos e Técnicas,” (Tutorial) In: II Congresso Brasileiro de Software: Teoria e Prática, 2011, São Paulo. Anais do II Congresso Brasileiro de Software: Teoria e Prática, 2011.

Bueno, P.M.S., Wong, W.E. e Jino, M., “Automatic Test Data Generation Using Particle Systems,” Proceedings of the ACM Symposium of Applied Computing (SAC 2008). ACM, New York, NY, USA, p. 809-814, 2008.

Bueno, P.M.S., Wong, W.E. e Jino, M., “Improving Random Test Sets Using the Diversity Oriented Test Data Generation,” Proc. of the Second International Workshop on Random Testing (RT 2007), ACM ASE, 2007.

Sumário

1. Capítulo 1 – Introdução.....	1
1.1. Contexto	1
1.2. Motivação e Objetivos	3
1.3. Organização	5
2. Capítulo 2 – Revisão de Conceitos.....	9
2.1. Definição, Objetivo e Limitações do Teste	9
2.2. Conceitos e Terminologia.....	12
2.3. Níveis do Teste de Software.....	17
2.4. Tipos de Teste de Software	18
2.5. Processo e Documentação do Teste	20
2.6. Técnicas e Critérios de Teste.....	22
2.6.1. Conceitos Iniciais.....	22
2.6.2. Técnicas e Critérios de Teste Baseados em Especificação e em Modelos	27
2.6.3. Técnica de Teste Baseada em Estrutura.....	37
2.6.4. Técnicas de Teste Baseadas em Defeitos	48
2.7. Modelos de Detecção de Defeitos	51
2.7.1. Domínios de Falha e Regiões de Falha.....	52
2.7.2. Continuidade de Regiões de Falha	54
2.8. Avaliação Empírica em Engenharia de Software.....	61
2.9. Considerações Finais	62
3. Capítulo 3 – Geração de Dados de Teste.....	63
3.1. Geração de Dados de Teste: Fundamentos e Trabalhos Iniciais	64
3.2. Geração de Dados de Teste por Meio de Execução Simbólica	67
3.3. Abordagem Dinâmica Para a Geração de Dados de Teste.....	69
3.4. Engenharia de Software Baseada em Busca.....	72
3.5. Meta-heurísticas	73
3.5.1. Subida de Encosta.....	75
3.5.2. Recozimento Simulado	76
3.5.3. Algoritmo Genético	78
3.5.4. Algoritmo de Seleção Clonal	83
3.5.5. Otimização por Enxame de Partículas	84
3.5.6. Outras Meta-heurísticas	86
3.6. Geração de Dados de Teste Baseada em Busca.....	86
3.7. Teste Baseado em Busca com Ênfase na Implementação	88

3.8. Teste Baseado em Buscas com Ênfase em Modelos e/ou na Especificação.....	98
3.9. Outras Abordagens de Teste Baseado em Busca.....	104
3.10. Pontos-chave e um Procedimento Genérico para Abordagens SBST	105
3.11. Considerações Finais	109
4. Capítulo 4 – A Diversidade Como uma Técnica de Teste de Software	113
4.1. Uma Introdução Sobre o Conceito de Diversidade	114
4.1.1. Diversidade nas Ciências Sociais – criatividade e inovação.....	115
4.1.2. Diversidade na Química – representatividade.....	116
4.1.3. Diversidade nas Ciências Biológicas – preservação e proteção	116
4.2. A Diversidade no Teste de Software	120
4.2.1. Uma Visão Intuitiva da Diversidade	120
4.2.2. Diversidade, Tolerância a Defeitos, Confiabilidade, Particionamento e Cobertura ...	126
4.3. Geração de Dados de Teste Orientada à Diversidade.....	128
4.3.1. A Diversidade Ideal sob a Ótica de Técnicas de Teste.....	129
4.3.2. Perspectivas e um Meta-Modelo Para a Diversidade.....	130
4.4. Perspectiva do Domínio de Entrada Para a Geração de Dados de Teste Orientada à Diversidade	133
4.4.1. Definições Iniciais	133
4.4.2. Distância e Diversidade.....	135
4.4.3. Domínio de Busca e Domínio de Entrada	140
4.5. Sumário e Propriedades de DOTS Para o Teste	143
4.6. Meta-heurísticas Para a Geração de Conjuntos de Teste Orientados à Diversidade (DOTS)	146
4.6.1. Recozimento Simulado Para Geração de DOTS	147
4.6.2. Algoritmo Genético para Geração de DOTS	149
4.6.3. Repulsão Simulada: uma Meta-heurística Auto-organizável para Otimizar Diversidade	153
4.7. Trabalhos Relacionados	161
4.7.1. Seleção de Dados de Teste por Análise e Agrupamento de Perfis de Execução	162
4.7.2. Geração de Sequências de Estados Baseada em Similaridade	163
4.7.3. Teste Antialeatório.....	165
4.7.4. Teste Aleatório Adaptativo.....	165
Teste Aleatório Adaptativo e Diversidade	168
4.7.5. Medida de Variabilidade do Teste com o uso de Distância de Informação.....	170
4.7.6. Utilização de Estados de Dados Escassos para Direcionar a Busca de Dados de Teste	171
4.8. Considerações Finais	172
5. Capítulo 5 – Avaliação Empírica da Geração de Dados de Teste Orientada à Diversidade	175
5.1. Desempenho das Meta-heurísticas para Gerar Diversidade.....	176
5.2. Simulação Monte Carlo para Avaliar Fatores que Influenciam a Eficácia da DOTG-ID	178

5.3. Resultados da Simulação	182
5.3.1. Relação entre a Eficácia dos Conjuntos DOTS e os Padrões de Falha	185
5.3.2. Relação entre a Eficácia dos Conjuntos DOTS e o Tamanho do Conjunto de Teste	187
5.3.3. Relação entre a Eficácia dos Conjuntos DOTS e o Tamanho das Regiões de Falha	189
5.3.4. Simulação: Sumário dos Resultados	190
5.4. Avaliação do Efeito da Diversidade na Cobertura de Fluxo de Dados e no Escore de Mutação	190
5.4.1. Procedimento de Avaliação	192
5.4.2. Programas e Drivers de Teste.....	194
5.4.3. Conteúdo e Formato dos Resultados	195
5.4.4. Análise Estatística.....	196
5.4.5. Resultados da Avaliação	199
5.4.6. Cobertura de Fluxo de Dados e Escore de Mutação: Sumário dos Resultados.....	205
5.5. Considerações Finais	208
6. Capítulo 6 – Conclusão	211
6.1. Resumo do trabalho	211
6.2. Contribuições.....	215
6.3. Trabalhos Futuros	217
Apêndice A – Técnicas de Teste: Definições e Detalhamentos.....	223
Teste Baseado em Modelos.....	229
Definição de Critérios da Técnica de Teste Baseada em Estrutura e descrição da Ferramenta Poke-Tool	245
Apêndice B – Geração de Dados de Teste: detalhamento das abordagens	257
Geração de dados de teste com uso de execução simbólica: outros trabalhos.....	257
Abordagem dinâmica para a geração de dados de teste: outros trabalhos.....	258
Teste Baseado em Buscas com Ênfase na Implementação.....	260
Teste Baseado em Buscas com Ênfase em Modelos e/ou na Especificação	266
Apêndice C – Detalhamento da Base Conceitual da Técnica DOTG	273
Diversidade e Software: Aumentando a Confiabilidade de Sistemas por Meio da Tolerância a Defeitos.....	273
O Conceito de Tolerância a Defeitos	273
Diversidade de Projeto.....	274
Diversidade de Dados	281
A Relação Entre a Diversidade no Teste e a Confiabilidade de Software.....	288
Modelos de Confiabilidade de Software	289
O Teste de Software para Avaliar a Confiabilidade	290
Impacto de Aspectos do Teste na Confiabilidade do Software.....	290
Diversidade do Teste e Confiabilidade de Software: Resumo	294
Técnicas de Teste Baseadas em Particionamento	296

Eficácia do Teste Baseado em Particionamento em um Contexto Determinista.....	298
Eficácia do Teste Baseado em Particionamento: a Influência da Incerteza	301
Sumário: Técnicas Baseadas em Particionamento e Diversidade.....	304
Diversidade no Sistema Imune	305
Memória e Capacidade de Generalização	305
Mecanismos de Promoção de Diversidade	306
Espaço de Formas, Afinidade e Reconhecimento Imune	306
A Estratégia de Cobertura do Sistema Imune.....	308
Resumo: um Paralelo Entre o Reconhecimento Imune e a Revelação de Defeitos	312
Apêndice D – Esboço da perspectiva de diversidade DOTG – Execution Information.....	315
Referências	321

Capítulo 1 – Introdução

Este capítulo apresenta a introdução desta tese. É apresentado o contexto do trabalho, são descritos aspectos que a motivaram, assim como os objetivos que nortearam o trabalho.

1.1. Contexto

Segundo Myers (1979), “teste é o processo de executar um programa com a intenção de encontrar defeitos”. Beizer (1990) destaca o teste como atividade de avaliação de comportamento: “teste de software é a execução do software de uma forma controlada com o objetivo de avaliar se o software se comporta ou não conforme especificado”. Uma definição mais recente (SWEBOK, 2004): “o teste de software consiste na verificação dinâmica do comportamento de um programa em relação ao comportamento esperado, com um conjunto finito de casos de teste, adequadamente selecionados a partir de um domínio de execuções usualmente infinito”.

O teste de software é uma atividade essencial para avaliar a qualidade do software, verificar a adequação do seu comportamento, e identificar as suas deficiências. Existem duas questões essenciais em teste de software: (a) é inviável avaliar o comportamento do software com todas as possíveis situações de entrada, fazendo com que seja necessário selecionar uma amostra de casos de teste a serem executados; (b) é desejável que esta amostra permita encontrar defeitos que existam e também resulte em uma avaliação abrangente e rigorosa do comportamento do software.

Técnicas e critérios de teste permitem a seleção e/ou avaliação de casos de teste. As técnicas de teste determinam a natureza da informação utilizada, por exemplo: teste baseado na especificação (Ostrand e Balcer, 1988), (Beizer, 1990), teste baseado na estrutura (Myers, 1979), (Rapps e Weyuker, 1985), (Maldonado, Chaim e Jino, 1992), ou teste baseado em defeitos (DeMillo, Lipton e Sayward, 1978), (White e Cohen, 1980), (Delamaro e Maldonado, 1996). A utilização de uma técnica de teste ocorre por meio da aplicação de um critério de teste, que estabelece os elementos requeridos a serem exercitados no teste (Goodenough e Gerhart, 1975), (Beizer, 1990).

A geração de dados de teste visa selecionar pontos do domínio de entrada do programa (dados de teste) para satisfazer um dado critério. Quando feita manualmente esta tarefa requer do testador conhecimentos específicos do critério utilizado, além da cuidadosa – e dispendiosa – análise da especificação do software, de modelos do software, ou do código do software em teste. Deste modo, a automação da geração de dados de teste é uma área de grande importância. Os primeiros trabalhos nesta área aplicavam a técnica de execução simbólica (Clarke, 1976), (Howden, 1975), (Boyer et al., 1975), (Ramamoorthy et al., 1976), ou a técnica de execução dinâmica (Miller e Spooner, 1976), (Korel, 1990), (Gallagher e Narasimhan, 1997).

Encontram-se na literatura trabalhos caracterizados pela aplicação de técnicas de busca para a solução de diferentes problemas de engenharia de software. Tais trabalhos reformulam problemas da engenharia de software como problemas de otimização baseada em busca, linha de pesquisa referida, a partir de 2001, como Engenharia de Software Baseada em Busca (Search Based Software Engineering – SBSE) (Harman e Jones, 2001). A área de trabalho SBSE tem avançado em diferentes frentes, como: priorização de requisitos em incrementos de software; otimização de métricas de qualidade de projeto; balanceamento de objetivos de qualidade de serviço; composição de serviços em arquiteturas orientadas a serviços; planejamento e alocação de recursos, dentre outras. Descrições de linhas de trabalho SBSE são apresentadas por Mantere e Alander (2005) e Harman et al. (2009). Vergilio et al., (2011) destacam trabalhos de autores brasileiros nesta área.

Atualmente, o repositório de trabalhos SBSE possui um total de 1020 artigos, sendo 540 trabalhos relacionados ao teste e depuração ¹. Isto mostra que a aplicação de técnicas de busca para problemas de teste de software tem sido priorizada e tem amadurecido de forma acentuada. Esta linha de trabalhos é referida como Teste de Software Baseado em Busca (Search Based Software Testing – SBST).

Pesquisadores da área de SBST têm focado prioritariamente na automação da geração de dados de teste para exercitar os requisitos de técnicas e critérios de teste (McMinn, 2004), (Harman et al., 2009), (McMinn, 2011). No teste baseado em estrutura, podem ser encontradas aplicações de SBST para exercitar ramos, nós ou caminhos em programas, ex: (Jones et al., 1996). No teste baseado em modelos, o foco principal é em modelos de estados (ex: EFSM – máquinas de estados finitos estendidas), e o objetivo é exercitar estados, transições de estados, ou caminhos nos modelos, exemplo: (Derderian et al., 2005). Outros trabalhos têm por objetivo a avaliação de requisitos não funcionais do software em teste (Afzal et al., 2009). Uma área de investigação recente considera situações em que o teste é realizado com mais de um objetivo, muitas vezes objetivos conflitantes (Lakhoria et al., 2007).

1.2. Motivação e Objetivos

Neste trabalho, é apresentada uma técnica de geração de dados de teste que tem como alvo propriedades de conjuntos de teste. Por propriedades entendem-se características descritas matematicamente, não necessariamente vinculadas a técnicas e critérios de teste existentes. A ideia é aproveitar o potencial de meta-heurísticas para gerar dados de teste que apresentem propriedades que, espera-se, resultem em um teste eficaz. Em relação a trabalhos anteriores em SBST, a diferença desta proposta é que as meta-heurísticas são utilizadas como uma ferramenta para explorar novas possibilidades em termos de técnicas e critérios de teste, e não apenas para gerar automaticamente dados para exercitar elementos requeridos de técnicas e critérios conhecidos.

¹ Repositório SBSE: http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/repository.html consultado em Julho de 2012.

Importante destacar que o problema da geração de dados de teste envolve espaços de busca multidimensionais e extensos, definidos pelo domínio de entrada do software em teste. As funções a serem otimizadas relacionam-se tipicamente à semântica do software, não sendo possível assumir características como a linearidade, ou a continuidade destas funções. Tais características fazem com que a utilização de técnicas robustas, como as meta-heurísticas, seja adequada.

Como propriedade alvo para a geração de dados de teste, foi escolhida a diversidade do conjunto de teste. A motivação para esta escolha foi a intuição de que a variabilidade (diversidade) de situações exploradas durante o teste tem um papel relevante para a completude e qualidade do teste realizado. Isto é, considerando o objetivo de testar o software de modo abrangente, a diversidade existente entre os dados de teste é uma propriedade a ser promovida. Outras propriedades de conjuntos de teste poderiam ser escolhidas; por exemplo, pela consideração de aspectos como o custo do teste, os riscos do software, a complexidade do software, etc.

O conceito de diversidade aparece de modo informal em guias de “boas práticas” e está embutido em heurísticas, como o teste exploratório (Bach, 2003), e também em técnicas baseadas em particionamento (Weyuker e Jeng, 1991). Contudo, a utilização do conceito de diversidade para gerar dados de teste não foi ainda abordada na literatura de teste de software.

O trabalho descrito nesta tese tem as seguintes características principais:

- Utilização de meta-heurísticas para gerar dados de teste que têm como alvo propriedades de conjuntos de teste, sendo que propriedades não são necessariamente vinculadas a técnicas e critérios de teste conhecidos. Dito de outro modo: propõe-se que as meta-heurísticas são ferramentas para o desenvolvimento de novas técnicas e critérios de teste, além de serem ferramentas para a geração de dados de teste.
- Tratamento do conceito de diversidade no teste de software e definição de medidas de diversidade de conjuntos de teste. Estas medidas são a base para a

geração de conjuntos de dados de teste de alta diversidade. Esta nova técnica para geração de dados de teste é referida como Geração de Dados de Teste Orientada à Diversidade (*Diversity Oriented Test Data Generation – DOTG*). Os conjuntos de dados de teste que têm a propriedade de apresentar alta diversidade são referenciados como Conjuntos de Teste Orientados à Diversidade (*Diversity Oriented Test Sets – DOTs*).

- Desenvolvimento de meta-heurísticas para geração de dados de teste orientada à diversidade. Foram consideradas três alternativas, a SA-DOTG, baseada em Recozimento Simulado; a GA-DOTG, baseada em Algoritmos Genéticos; e a meta-heurística Repulsão Simulada para geração de dados de teste (SR-DOTG). A SR-DOTG, proposta nesta tese, é baseada na simulação de sistemas de partículas (dados de teste) sujeitas a forças magnéticas de interação.

1.3. Organização

O conteúdo restante desta tese está organizado em seis capítulos e quatro apêndices, conforme descrito a seguir.

O Capítulo 2 apresenta os fundamentos da atividade de teste de software, com ênfase em técnicas e critérios de teste. Aborda, entre outros pontos: definição, objetivo e limitações do teste; conceitos e terminologia; níveis do teste de software; tipos de teste de software; processo e documentação do teste; técnicas e critérios de teste, abrangendo técnicas e critérios de teste baseados em especificação e em modelos; técnicas de teste baseadas em estrutura; e técnicas de teste baseadas em defeitos. Aborda também conceitos relacionados aos mecanismos da revelação de defeitos e aspectos de avaliação empírica em engenharia de software.

O Capítulo 3 descreve as abordagens para a geração de dados de teste: a execução simbólica e a técnica dinâmica. Descreve a Engenharia de Software Baseada em Buscas – SBSE e o Teste de Software Baseado em Buscas – SBST. Aborda o conceito de meta-heurística e descreve algumas delas. Fornece uma visão geral de como um engenheiro de

software pode definir uma abordagem SBST e descreve pontos-chave a serem considerados nesta definição. Descreve então trabalhos em SBST, com ênfase no teste baseado em implementação e no teste baseado em modelos.

O Capítulo 4 apresenta a técnica Geração de Dados de Teste Orientada à Diversidade (*Diversity Oriented Test Data Generation – DOTG*) e as meta-heurísticas para a geração automática de dados de teste para esta técnica. É apresentado o conceito de perspectivas para a diversidade e um “meta-modelo” para a diversidade, que fornece um guia para o desenvolvimento de perspectivas da DOTG. É definida a perspectiva do domínio de entrada para a Geração de Dados de Teste Orientada à Diversidade (DOTG-ID). As características dos conjuntos de teste de alta diversidade (DOTS), gerados com a perspectiva DOTG-ID, são analisadas sob a ótica de modelos de detecção de defeitos e de modelos de caracterização de domínios de falha e regiões de falha. São detalhadas as meta-heurísticas implementadas para a geração automática de dados de teste de alta diversidade (DOTS). O capítulo é finalizado com a descrição de trabalhos que apresentam algum aspecto em comum com a técnica proposta DOTG.

O Capítulo 5 apresenta as avaliações empíricas realizadas para analisar as características da técnica de geração de dados de teste proposta (DOTG-ID) e das meta-heurísticas implementadas. É descrita a avaliação da eficiência e eficácia das meta-heurísticas para gerar conjuntos DOTS em diferentes situações. É apresentada a simulação Monte Carlo realizada com o objetivo de estudar a influência dos fatores: tipo de região de falha; taxa de falhas; e tamanho do conjunto de teste na eficácia da técnica DOTG-ID. É também descrito o experimento realizado para avaliar o efeito da diversidade dos conjuntos de teste na cobertura alcançada no teste de programas, medida considerando critérios de teste baseados em análise de fluxos de dados e no critério baseado em defeitos Análise de Mutantes.

O Capítulo 6 apresenta um resumo do trabalho, destaca as contribuições realizadas e aponta os trabalhos futuros.

O Apêndice A apresenta um conteúdo complementar ao Capítulo 2. São abordados o Teste Combinatório e o Teste Baseado em Modelos e são descritos critérios de teste

baseados em estrutura e a ferramenta POKE-TOOL, assim como o teste baseado em análise de mutantes e a ferramenta Proteum.

O Apêndice B complementa o Capítulo 3. São descritos outros trabalhos que aplicam a execução simbólica e a execução dinâmica para a geração de dados de teste. São também fornecidos detalhes adicionais dos trabalhos na área SBST.

O Apêndice C oferece uma descrição mais aprofundada de trabalhos que formaram a base conceitual para a proposta da técnica DOTG, apresentada no Capítulo 4. O conteúdo presente neste apêndice é também descrito de forma resumida no Capítulo 4.

No Apêndice D, é apresentado um esboço da perspectiva da diversidade que leva em conta informações sobre a execução do software durante o teste (DOTG-EI), trabalho futuro proposto no Capítulo 6.

Capítulo 2 – Revisão de Conceitos

“No amount of experimentation can ever prove me right; a single experiment can prove me wrong.”

Albert Einstein

Este capítulo apresenta os fundamentos da atividade de teste de software, com ênfase em técnicas e critérios de teste. É apresentada a terminologia da área e são abordados conceitos como: níveis de teste, tipos de teste e processo de teste. O objetivo principal deste capítulo é fornecer uma base para a compreensão das várias técnicas e critérios de teste existentes na literatura e utilizados por profissionais do teste de software. São apresentados também alguns conceitos importantes para o entendimento da técnica de teste proposta nesta tese, como: regiões de falha, mecanismos associados à revelação e defeitos no teste, e avaliação empírica em engenharia de software.

O Apêndice A complementa o conteúdo deste capítulo, com uma descrição mais aprofundada de algumas técnicas, critérios e ferramentas de apoio ao teste.

2.1. Definição, Objetivo e Limitações do Teste

O significado do termo “teste”, segundo o dicionário Aurélio é: “exame, verificação ou prova para determinar a qualidade, a natureza ou o comportamento de alguma coisa, ou

de um sistema sob certas condições”. O termo “testar” significa: “submeter a teste ou experiência; fazer funcionar experimentalmente, provar, experimentar”.

Testar um software pode ser entendido informalmente, portanto, como o ato de experimentar o software e verificar o seu comportamento. Existem diversas definições de teste de software; por exemplo:

- “Teste é uma atividade direcionada para avaliar um atributo ou capacidade de um programa ou sistema e determinar se ele satisfaz os resultados requeridos” (Hetzel, 1988);
- “Teste é o processo de executar um programa com a intenção de encontrar defeitos” (Myers, 1979);
- “Teste é o processo pelo qual se explora e se entende o estado dos benefícios e riscos associados com a versão de um sistema de software” (Kaner et al., 2002).
- “Teste de software consiste na verificação dinâmica do comportamento de um programa, através de um conjunto finito de casos de teste, adequadamente selecionado a partir de um conjunto infinito de possibilidades, contra um comportamento esperado especificado” (SWEBOK, 2004);
- “Teste de software é a execução do software de uma forma controlada com o objetivo de avaliar se o software se comporta ou não conforme especificado” (Beizer, 1990).

Por meio dessas definições é possível compreender objetivos para a realização do teste de software:

- Avaliar se o software produzido atende aos requisitos levantados com os usuários do software. Por exemplo, avaliar se as funcionalidades previstas estão presentes no software e se elas apresentam o comportamento esperado; avaliar se o desempenho do software (tempo de resposta) é aceitável; ou avaliar se os padrões para as interfaces de usuário foram respeitados.

- Identificar deficiências que podem existir em produtos de software. Revelar situações em que o software não se comporta como o esperado, ou produz resultados incorretos, diferentes dos esperados.
- Obter evidências da qualidade do software. Ao se testar o software de forma sistemática e rigorosa, e ao se observar que ele apresenta o comportamento previsto e produz os resultados esperados, ganha-se confiança de que o software apresenta um nível suficiente de qualidade. Notar que “qualidade” neste contexto refere-se genericamente a características desejáveis em software como: correção, facilidade de uso, eficiência, confiabilidade, etc. Notar também que o que é um “nível suficiente de qualidade” de um software pode variar enormemente dependendo do seu propósito e domínio de aplicação.

Uma limitação do teste de software é que é impossível, em geral, demonstrar que o software é correto por meio de teste. Segundo Dijkstra “o teste pode apenas mostrar a presença de um erro; o teste nunca pode provar a ausência de todos dos erros” (Dijkstra, 1972). Apenas o teste exaustivo, no qual todas as possíveis combinações de entrada são executadas, poderia garantir esta correção de funcionamento (Goodenough, e Gerhart, 1975). Na grande maioria dos casos o teste exaustivo não é factível e o número de situações em que o software é testado (número de cenários de teste) representa uma pequena fração do número total de situações possíveis.

Por exemplo, considere um software que tenha como entrada 15 variáveis, sendo que cada uma delas pode assumir 7 valores diferentes. Considere ainda que a execução de cada teste demore 1/100 de segundo (0,01 segundo). Para realizar o teste exaustivo deste software seriam necessários $7^{15}/100$ segundos, o que equivale a 1.505 anos e alguns meses de processamento.

Neste sentido, o teste é análogo a um contraexemplo de um teorema. Basta um único caso em que o teorema não funciona para prová-lo incorreto, mas vários casos em que o teorema funciona não provam que ele é verdadeiro.

Esta natureza inerentemente incompleta de qualquer teste impõe o desafio de se buscar o máximo possível de eficácia, dentro do que for factível em termos de esforço, tendo como baliza o tipo de aplicação e os requisitos de qualidade do software. Um *teste eficaz* é aquele que permite revelar a maioria dos defeitos existentes no software, idealmente revelar todos os defeitos (Myers, 1979).

A impossibilidade do teste completo implica em uma incerteza sobre qualquer afirmação sobre a qualidade do software submetido ao teste. Não se pode estar certo que um software testado tenha boa qualidade, mesmo se forem revelados poucos defeitos (ou nenhum defeito). Mas esta incerteza diminui na medida em que aumentam as evidências de que um teste sistemático e cuidadoso foi realizado. Isto é, evidências de que um teste feito foi eficaz aumentam a confiança de que o software testado (e que apresentou defeitos) é realmente um bom produto. No outro extremo, se o teste de um software foi realizado de forma improvisada, sem o emprego de boas práticas, pouco se pode afirmar sobre a qualidade do software testado, independentemente do número de defeitos revelados.

Um dos objetivos do desenvolvimento é produzir um software correto. O *paradoxo do teste* é que encontrar defeitos destrói a confiança de que o software está correto. Seria então o propósito do teste destruir a confiança no software, ao invés de construí-la? A questão, conforme abordado nos parágrafos anteriores, é que a melhor maneira de ganhar confiança no software é tentar destruí-la (a confiança) de forma sistemática e rigorosa (Beizer, 1990).

2.2. Conceitos e Terminologia

Uma falha de software ocorre por meio de um processo complexo que pode ocorrer na execução do software. Três conceitos são importantes para compreensão do processo de falha: *falha*, *defeito*, e *erro*. Uma *falha* (*failure* em inglês) é a ocorrência de uma discrepância entre o resultado observado da execução do software e o resultado prescrito pelos requisitos. Um *defeito* (*fault* ou *defect*) é uma anomalia no software que pode causar uma falha (exemplos: um comando imperfeito, incompleto, ausente ou extra no software).

Um *erro* (*error*) é um estado incorreto, intermediário ou final, de execução do software que pode produzir um resultado incorreto, ou seja, pode levar a uma falha do software. Um defeito no software é introduzido devido a algum erro humano ou *engano* (*mistake*) cometido em algum momento no desenvolvimento do software e não descoberto em atividades de inspeção. Na literatura é comum que a palavra *erro* seja empregada em contextos menos rigorosos para se referir aos conceitos: *falha*, *defeito*, *erro* e *engano*. O termo “*bug*” é também usado para referir-se a um defeito (IEEE Std 610, 1990).

Um caso clássico presente na literatura refere-se a um acidente com o foguete Ariane V, que explodiu em 1996, cerca de 40 segundos após ter decolado (Lions, 1996). O foguete saiu de sua rota programada e se desintegrou devido a forças aerodinâmicas. Em uma análise posterior do incidente, verificou-se que o software de controle do voo indicou uma direção errada ao foguete (esta foi a falha) devido a uma conversão incorreta de uma variável tipo real de 64 bits em um inteiro de 16 bits (este foi o defeito). Esta conversão produziu um erro de overflow, isto é, um valor errôneo de uma variável (este foi o erro), que ocasionou a resposta incorreta do software. O engano foi, provavelmente, cometido por um programador, que inseriu no software um comando de atribuição indevido.

Portanto, para um defeito ser descoberto é necessário que o software seja executado de uma forma tal que o defeito seja “sensibilizado” e provoque uma falha. Isto ocorre quando o comando defeituoso é executado e cria um estado de erro, por exemplo, valores incorretos para as variáveis internas, ou uma avaliação incorreta de alguma condição. Para que a falha aconteça, é necessário ainda que este erro seja propagado, ao longo da execução, do ponto onde foi criado (logo após o comando defeituoso) para alguma saída do software, ocasionando uma resposta final incorreta (valores incorretos para alguma saída produzida, por exemplo). A Figura 1 ilustra a cadeia de eventos que se inicia com um engano e que pode resultar em uma falha do software. Modelos sobre o processo de falhas em software são descritos em Ehrlich et al (1991), Kuhn et al (2004), Thompson et al (1993) e Voas (1992).

Quando ocorre uma falha do software durante o teste, normalmente ocorre a atividade de *depuração* do software. A *depuração* é distinta do teste, embora sejam atividades

diretamente relacionadas. A depuração baseia-se em informações sobre as circunstâncias da falha do software – oriundas do teste, com o objetivo de identificar o defeito que causou a falha e remove-lo (Sommerville, 2006).



Figura 1. Cadeia de Eventos Para a Falha do Software

Variáveis de entrada de um software são aquelas que recebem os valores a serem utilizados no processamento, por exemplo, em parâmetros passados ao software, ou recebidos em algum campo de uma interface de usuário (Korel, 1990). *Variáveis de saída* são aquelas que fornecem os resultados do processamento, por exemplo, em parâmetros passados a outros componentes, ou informados pela interface de usuário. Cada variável de entrada é de um tipo específico (exemplos: inteiro, real, caractere, vetor, registro) e possui um *domínio*, que é definido como todos os possíveis valores que a variável pode assumir (por exemplo, na linguagem C, em uma representação de 32 bits, uma variável do tipo inteiro pode assumir valores de - 2,147,483,647 a + 2,147,483,647). Comumente são associadas às variáveis de entrada uma faixa de valores, ou um conjunto de valores, que podem ser atribuídos à variável (por exemplo, variável idade, tipo *inteiro*, entre 0 e 130). Variáveis de saída também são associadas a tipos específicos, mas as faixas ou conjuntos de valores de saída dependem da computação específica realizada pelo software. Naturalmente existem variáveis internas do software, que não são de entrada nem de saída.

O *domínio de entrada* (DE) do software (S) é o conjunto de todos os possíveis valores que podem ser utilizados para executar o software. Esse domínio é definido pela combinação dos domínios de todas as variáveis de entrada do software. Analogamente, o *domínio de saída* (DS) é definido pela combinação de valores para as variáveis de saída do software.

Considere, por exemplo, um software que recebe como parâmetros de entrada dois valores do tipo inteiro (x , y), positivos e menores que 100, e produz como saída o resultado

(r) que é o primeiro valor recebido (x) elevado ao segundo valor (y). Caso os parâmetros recebidos tenham valores diferentes dos aceitos, uma mensagem é emitida.

Neste caso as variáveis de entrada são x e y , com faixas de valores $0 \leq x \leq 100$ e $0 \leq y \leq 100$. DE consiste de todas as possíveis combinações x , y , sendo portando um domínio bidimensional. A cardinalidade do domínio de entrada ($|DE|$) é definida pelo máximo valor passível de representação para variáveis do tipo inteiro (negativos e positivos) e, tratando-se de um espaço bidimensional, esta cardinalidade é obtida elevando este valor máximo ao quadrado.

Neste exemplo o domínio de saída é formado por uma variável (r), que pode assumir valores entre 1 e 100^{100} , e um variável tipo vetor de caracteres, que pode ter o conteúdo “valores de entrada incorretos” (caso x ou y estejam fora das faixas aceitas) ou conteúdo nulo, caso contrário. Notar que no domínio de entrada DE há duas classes de elementos distintos: elementos “válidos”, que resultam em valores para a variável r como saída ($0 \leq x \leq 100$ e $0 \leq y \leq 100$); e elementos “inválidos”, que resultam na mensagem “valores de entrada incorretos” ($x < 0$, ou $x > 100$, ou $y < 0$, ou $y > 100$).

Essas definições de *variáveis de entrada*, *variáveis de saída*, *domínio de entrada* e *domínio de saída* são simplificações bem aceitas que representam uma boa parte do software existente. Entretanto, os conceitos de variáveis e domínios podem requerer alguma adaptação para situações mais particulares, por exemplo:

- Software que recebe como entrada (ou produz como saída) um software (ex: compiladores);
- Software que recebe como entrada linguagem natural (ex: mecanismos de busca);
- Software que recebe como entrada (ou produz como saída) imagens (ex: software de processamento de imagens, software de reconhecimento de imagens);
- Software que recebe como entrada uma sequencia eventos e que realiza atualizações de estado com base em cada evento recebido da sequencia (ex: software embarcado)

Notar que em uma perspectiva ampla o domínio de entrada deve considerar qualquer aspecto que potencialmente afete a execução do software. Exemplos: o estado específico da base de dados antes da execução; os valores das variáveis globais utilizadas pelo software; ou aspectos de ambiente como o sistema operacional e as configurações específicas de operação. Um tratamento tecnicamente adequado, segundo Hamlet (2006) é forçar que o estado da execução atinja um estado inicial desejado e então, depois de alcançado o estado inicial, as entradas e ações do teste devem ser fornecidas. Offutt e Abdurazik (1999) chamam de *prefixo do caso de teste* as entradas necessárias para colocar o software em um estado apropriado para a execução de um teste.

Um *dado de teste* (ou dado de entrada) é um conjunto de valores atribuídos para as variáveis de entrada e que provoca uma única execução do software. Um *caso de teste*, além de valores atribuídos para as variáveis de entrada, também define o *resultado esperado* (re) do software, isto é, a saída a ser obtida se o software estiver correto. Um *conjunto de dados de teste* T é uma coleção finita de dados de teste, $|T|$ é o tamanho do conjunto de dados de teste (IEEE Std 610).

No exemplo anterior (software que eleva x a y e fornece o resultado) um dado de teste é um par de valores para x e y (ex, $x = 8$, $y = 5$). O resultado esperado é o valor correto para r , ou a mensagem de erro, se x ou y estejam fora das faixas aceitas. Um caso de teste exemplo seria $x = 8$, $y = 5$, $re = 32768$.

Uma premissa importante do teste é que haja algum mecanismo que permita avaliar se um resultado ou comportamento produzido pelo software está de acordo com a especificação; ou seja, se é o resultado esperado de um software correto. Este mecanismo, chamado *oráculo*, é normalmente uma comparação manual “esperado versus obtido” que identifica se o software falhou ou não (Peters e Parnas, 1998).

Para alguns tipos de software é difícil determinar se o resultado produzido está correto. Isto ocorre, por exemplo, se um software deve fornecer como saída o resultado de uma função matemática muito complexa (Weyuker, 1982). Quando não há um oráculo possível para testar o software, alternativas podem ser consideradas, tais como: utilizar dados de teste específicos para os quais se conhece o resultado esperado; utilizar duas

implementações diferentes (duas versões) da mesma especificação e comparar resultados produzidos para cada versão; ou aceitar resultados plausíveis assumindo-os como corretos. Quando não há um oráculo, portanto, há uma considerável chance de que uma falha do software em teste não seja percebida como tal.

2.3. Níveis do Teste de Software

O teste de software é normalmente realizado em uma série de fases, ou níveis: Teste de Unidade, Teste de Integração, Teste de Sistema e Teste de Aceitação (Sommerville, 2006). Em cada nível as características e natureza do teste realizado são diferentes, assim como os tipos de defeitos encontrados. Dependendo do sistema a ser testado, outros níveis de teste podem ocorrer, por exemplo, quando software, dispositivos mecânicos e hardware específicos precisam ser integrados e testados.

Após o desenvolvimento de cada unidade do software (procedimento, função, método ou classe) é realizado o *teste de unidade* (ou *teste unitário*), que visa a identificar defeitos introduzidos nos algoritmos e estruturas de dados dessas unidades. Em geral, o teste de unidade é feito pelo próprio desenvolvedor da unidade. As unidades são então incrementalmente integradas e testadas (*teste de integração*). Esta etapa é realizada pelos desenvolvedores ou por elementos de uma equipe de teste e visa a identificar defeitos de interface entre as unidades. Depois de integrado, o software é testado “como um todo”: o *teste de sistema* é o nível de teste cujos requisitos são derivados da especificação de requisitos funcionais e não funcionais, e é aplicado para verificar se o software e o hardware executam corretamente ou não quando integrados ao ambiente de operação. O *teste de aceitação* é então conduzido para estabelecer se o sistema satisfaz ou não os critérios de aceitação definidos com o cliente.

Deve-se notar que em cada um desses níveis (unidade, integração, sistema e aceitação) diversos *ciclos de teste* podem ocorrer. Em cada ciclo de teste um *procedimento de teste* é realizado, por meio do qual um ou mais casos de teste são executados. Falhas

provocadas por casos de teste devem ser registradas para posterior *depuração* do software pela equipe de desenvolvimento.

O teste de um software já testado previamente e que sofreu mudanças é chamado de *Teste de Regressão*. Este teste busca verificar se as modificações efetuadas não introduziram novos defeitos ou ativaram defeitos em partes inalteradas do código; visa também avaliar se o software ainda satisfaz os seus requisitos. O teste de regressão é tipicamente realizado na fase de manutenção, após a realização de mudanças no software ou no seu ambiente. Conforme descrito no parágrafo anterior, este teste também pode ocorrer ao longo do desenvolvimento do software.

2.4. Tipos de Teste de Software

Diversos atributos de qualidade do software podem ser avaliados por meio de testes. Características de qualidade de software definidas em padrões como a ISO/IEC 9126 (2001) podem servir como base para a definição de aspectos do software a serem avaliados por meio da realização de testes. Portanto, existem diversos *tipos de teste*; cada tipo visa à avaliação de um aspecto distinto do software e pode ser aplicado em um ou mais níveis de teste (unidade, integração, etc.). Exemplos de tipos de teste segundo Kaner et al (2002):

O *teste de funcionalidade* busca avaliar se o software apresenta ou não um conjunto de funções que satisfaça às necessidades levantadas (associadas aos requisitos definidos). Avalia também se o software produz saídas corretas.

O *teste de desempenho* visa a avaliar se o software executa as funções previstas satisfazendo ou não os requisitos de desempenho definidos (e.g., velocidade de processamento, tempo de resposta ou uso de memória).

O *teste de interoperabilidade* avalia a capacidade do software em interagir com um ou mais componentes ou sistemas.

O *teste de segurança (security)* visa a avaliar a habilidade do software de impedir o acesso não autorizado – acidental ou deliberado – ao software ou a dados.

O *teste de carga* consiste em medir o comportamento de um componente ou sistema submetido a cargas de processamento crescentes para determinar qual nível de carga o sistema pode tratar; por exemplo, número de usuários ou número de transações.

O *teste de estresse* simula condições atípicas de utilização do software, provocando aumentos e reduções sucessivas de transações que superem os volumes máximos previstos para a capacidade de processamento dos componentes ou do sistema.

O *teste de usabilidade* visa a determinar quão facilmente o produto de software é compreendido, apreendido e usado, e quão agradável ele é ao usuário.

O *teste de portabilidade* busca avaliar o nível de facilidade de transferência de um ambiente de hardware e software para outro ambiente.

O *teste de instalação* busca avaliar se o software é instalado e desinstalado com sucesso ou não em determinado ambiente operacional.

O *teste de recuperação* determina a capacidade ou incapacidade de um produto de software de restabelecer condições normais de operação e de recuperar dados afetados, em caso de falha.

A determinação de quais tipos de teste devem ser utilizados e em qual (quais) nível (níveis) eles devem ser aplicados, é feita considerando-se as informações disponíveis sobre a natureza do software em teste e sobre os requisitos específicos dos módulos que compõem o software. Tipicamente os testes que avaliam características não funcionais (desempenho, segurança, usabilidade, carga, etc.) são realizados na fase de teste de sistemas. Apesar disto, dependendo da natureza do software pode ser necessário submeter unidades e conjuntos integrados de unidades a esses testes. Exemplos: avaliar o desempenho de unidades críticas em software de tempo real, ou avaliar a segurança de um conjunto integrado de unidades de um software bancário.

Na maior parte das situações é adequado focar atenção no teste de funcionalidade para avaliar se os requisitos funcionais do software são satisfeitos ou não, e revelar defeitos de funcionalidade. Este teste deve ser complementado por outros tipos de teste para avaliar se os requisitos não funcionais do software são satisfeitos ou não.

2.5. Processo e Documentação do Teste

Um processo de software pode ser entendido como um “conjunto de atividades e resultados associados que geram um produto de software” (Sommerville, 2006), ou “um processo é um conjunto de passos parcialmente ordenados, constituídos por atividades, métodos, práticas e transformações, usado para atingir uma meta” (Padua Filho, 2003). Um processo é definido quando há uma documentação que descreve o que é feito (produtos), quando (passos) por quem (agentes), pelo que é usado (insumos ou recursos) e o que é produzido (Padua Filho, 2003).

A adoção de um processo coerente para realizar as diversas atividades da engenharia de software contribui positivamente para que se alcance o objetivo pretendido, seja em termos de cumprimento de cronograma e orçamento, ou em termos de qualidade dos artefatos gerados e do produto final.

É possível identificar na literatura modelos de processos genéricos de teste, que definem um conteúdo amplo em teste (formado por atividades, técnicas, artefatos, etc.) e que servem como base para a definição de processos customizados, fundamentados em necessidades de teste específicas de cada organização (Londesbrough, 2008), (Mette e Hass, 2008).

Um exemplo de modelo de processo genérico de teste é baseado em artefatos de teste determinados na norma IEEE Std 829 (1999), e é composto pelos subprocessos: Planejamento; Projeto, Execução e Registro; Acompanhamento; e Finalização. Cada um desses subprocessos tem um propósito específico e é composto por uma sequência de atividades, definidas de forma tal que o propósito do subprocesso seja atingido. As diversas atividades criam ou utilizam os artefatos que documentam o teste (plano de teste, projetos de casos de teste, procedimentos de teste, etc) – (Argollo et al., 2006), (Bueno et al., 2006), (Bueno et al., 2008).

O processo de teste deve ser acomodado no processo de software como um todo. Uma alternativa é organizar os subprocessos de teste segundo o Modelo V, que considera as principais fases do processo de software, associando a cada fase o nível de teste de

software correspondente – unidades, integração, sistema e aceitação (Padua Filho, 2003). As atividades de teste podem ser iniciadas junto com as atividades de desenvolvimento, logo que as informações necessárias estejam disponíveis. Neste caso, o planejamento do teste deve ser realizado junto com o planejamento do desenvolvimento, os projetos do teste de sistema e do teste de aceitação podem ser iniciados logo que os requisitos do software tenham sido definidos e podem ser refinados à medida que a modelos do software tenham sido desenvolvidos (fase de projeto do software). O projeto do teste de integração pode ocorrer quando informações sobre a arquitetura estejam elaboradas (fase de projeto do software).

Todos os projetos de teste de integração desenvolvidos são utilizados na execução do teste de integração. A execução ocorre assim que etapas de codificação tenham sido finalizadas e que as unidades estejam disponíveis para a integração do software. Após esta integração os projetos de teste de sistema e de aceitação são utilizados na execução desses testes.

Alguns modelos de processo de desenvolvimento de software, como o Espiral ou o RUP, definem iterações (ou ciclos) para o desenvolvimento do software, com incrementos (ou releases) associados a cada iteração (Pressman, 2006), (Sommerville, 2006). Nestes casos, as atividades de teste devem ser associadas às atividades de desenvolvimento em cada iteração, de modo que haja o adequado replanejamento, projeto, execução e registro, e acompanhamento dos testes. Recomenda-se realizar um planejamento global do teste junto com o planejamento do desenvolvimento. O teste final do sistema e o teste de aceitação são executados após o término da última iteração do processo de desenvolvimento.

Outros modelos enfatizam o projeto de testes e sua execução como um elemento central do desenvolvimento do software. A abordagem *Desenvolvimento Baseado em Testes* (ou *TDD*, sigla em inglês) é frequentemente adotada em métodos ágeis, como a *Programação Extrema* (XP). No *TDD* os casos de teste são criados pelos desenvolvedores antes mesmo que o código fonte seja criado (Erdogmus et al., 2005).

2.6. Técnicas e Critérios de Teste

Técnicas e critérios de teste são essenciais para a seleção ou avaliação de casos de teste a serem utilizados para se testar um produto de software. Técnicas e critérios estabelecem requisitos (restrições) a serem satisfeitos pelo teste e podem ser aplicadas nos diversos níveis de teste (unidade, integração, sistema) (Beizer, 1990). Embora existam técnicas e critérios direcionados a avaliar requisitos não funcionais do software (desempenho, segurança, usabilidade, etc.), a maior parte deles trata do teste de funcionalidade, isto é, buscam avaliar se o software apresenta ou não um conjunto de funções e características que satisfaçam as necessidades levantadas, além de identificar os defeitos existentes. Esta seção aborda técnicas e critérios para o teste de funcionalidade de software.

2.6.1. Conceitos Iniciais

Uma tarefa fundamental na atividade de teste é a seleção dos casos de teste a serem utilizados. Esta seleção é necessária porque, conforme já destacado, o *Teste Exaustivo*, feito com todas as possíveis combinações de valores de entrada, é quase sempre inviável. Tendo em vista o objetivo principal do teste de revelar defeitos e a restrição de que os cronogramas e orçamentos sejam cumpridos, é importante a utilização de *Técnicas de Teste* (Beizer, 1990), (Goodenough e Gerhart, 1975), (Whittaker, 2000), (Zhu e Hall, 1993). Estas técnicas estabelecem procedimentos para selecionar os casos de teste que, espera-se, resultem em um conjunto de dados de teste eficaz e de tamanho moderado. Técnicas de teste também fornecem meios para se avaliar um conjunto de dados de teste previamente selecionado, além de embasar decisões sobre quando o teste é considerado suficiente.

As técnicas de teste determinam a natureza da informação a ser utilizada para se fazer a seleção ou a avaliação de conjuntos de teste. Diversas classificações de técnicas de teste são encontradas na literatura e não existe consenso neste tema. No entanto, não é mandatório que exista uma classificação consensual e absoluta, desde que os conceitos estejam bem estabelecidos para embasar os projetistas de teste no seu trabalho de selecionar

e utilizar as técnicas, e os pesquisadores na análise das técnicas de teste existentes e no desenvolvimento de novas técnicas. Uma classificação tradicional define as seguintes técnicas: *Teste Funcional*, também chamado *Caixa Preta*; *Teste Estrutural* (*Caixa Branca*); e *Teste Baseado em Defeitos*.

Uma categorização atual, desenvolvida no contexto da norma ISO/IEC 29119 (2011), define as seguintes técnicas de teste: Técnicas Baseadas em Especificação (referida tradicionalmente como Teste Funcional); Técnicas Baseadas em Estrutura (referida tradicionalmente como Teste Estrutural); Técnicas Baseadas em Defeitos; e Técnicas Baseadas em Experiência.

Na *Técnica Baseada em Especificação* os *casos de teste* avaliam alguma expectativa sobre o software em teste e são selecionados por meio da análise da especificação do software ou de artefatos gerados ao longo do desenvolvimento do software (análise e projeto) – *modelos do software* (alguns autores consideram o teste com o uso de modelos do software como sendo uma classe distinta – o *Teste Baseado em Modelos*). Pode ser considerado que a *Técnica Baseada em Especificação* inclui também o *Teste Aleatório*, no qual os *dados de teste* são selecionados aleatoriamente do domínio de entrada do software e os *resultados esperados* são derivados da especificação ou de modelos do software (alguns autores consideram o Teste Aleatório como uma classe distinta).

Na *Técnica Baseada em Estrutura* os *dados de teste* são selecionados por meio da análise da estrutura interna do software e os *resultados esperados* são derivados da especificação ou de modelos do software (alguns autores se referem ao teste que busca exercitar a estrutura de *Modelos* também como teste estrutural).

Na *Técnica Baseada em Defeitos* *dados de teste* são selecionados a partir de classes de defeitos que podem ser introduzidos ao longo do desenvolvimento do software e os *resultados esperados* são derivados da especificação ou de modelos do software.

A *Técnica Baseada em Experiência* não utiliza descrições explícitas sobre o software em teste ou sobre o seu código. Os casos de teste baseiam-se na experiência sobre defeitos

comuns e/ou sobre a experiência adquirida pelo testador no uso do software em teste. O Teste Exploratório é um exemplo de Teste Baseado em Experiência (Bach, 2005).

As diversas técnicas de teste não são excludentes; na verdade, a utilização de várias técnicas é recomendada, pois os defeitos revelados pela aplicação de cada uma delas podem ser diferentes (Basili e Selby, 1987), (Maldonado, 1991), (Littlewood, 2000). Conforme já destacado anteriormente, a definição de qual (ou quais) técnica(s) deve (devem) ser utilizada(s) deve ser feita na fase de planejamento do teste e deve levar em conta o domínio de aplicação, a natureza do software, os objetivos de negócio e os riscos envolvidos.

A utilização de uma técnica de teste ocorre por meio da aplicação de algum *critério de teste* (Beizer, 1990) (Frankl e Weyuker, 1988), (Goodenough e Gerhart, 1975), (Maldonado, Chaim e Jino, 1992), (Myers, 1979), (Rapps e Weyuker, 1985), (Zhu e Hall, 1993). As técnicas de teste determinam diretrizes mais genéricas para o teste, enquanto os critérios associados às técnicas detalham especificamente as condições a serem satisfeitas no teste quando o critério é utilizado. Por exemplo, o critério de teste Todos os Comandos o qual requer que cada comando do software seja exercitado pelo menos uma vez, é um dos critérios da Técnica Baseada em Estrutura. Os conceitos *técnicas de teste* e *critérios de teste* muitas vezes são utilizados como sendo equivalentes em alguns artigos e normas técnicas.

Em geral, os critérios de teste podem ser utilizados para três propósitos distintos, embora relacionados (notar que nem todas as técnicas e critérios de teste descritos anteriormente podem ser utilizados para todos os propósitos):

- Criar um conjunto de casos de teste: direcionam a seleção de casos de teste que buscam exercitar aspectos específicos do software em teste;
- Avaliar um conjunto de casos de teste já criado: permitem avaliar se os casos de teste selecionados exercitam (ou não) aspectos específicos do software em teste, ou ainda;

- Servir como uma condição para a finalização do teste: definem uma maneira de se julgar se o teste de um software foi o suficiente ou se é necessário realizar mais testes.

Um critério de teste determina um conjunto de *Elementos Requeridos* do software que devem ser exercitados pela realização do teste. Por exemplo, o critério já mencionado Todos os Comandos (ou Todos os Nós) da técnica de teste Baseada em Estrutura, define como elementos requeridos os comandos do código do software. Neste caso, cada comando do software deve ser executado (ou *exercitado*), pelo menos uma vez, por algum dado de teste.

A *Cobertura do Teste*, com respeito a um critério de teste específico, mede o percentual de elementos requeridos exercitados por um conjunto de casos de teste executados. Por exemplo, o percentual de comandos executados durante o teste seria uma medida de cobertura considerando-se o critério Todos os Comandos. Deste modo, a medida de cobertura quantifica, sob certa perspectiva, a qualidade do teste realizado. É razoável supor, neste exemplo, que um conjunto de teste que executa 90% dos comandos de um software é melhor do que outro conjunto de teste que executa apenas 50% dos comandos. Se um conjunto de casos de teste exercita todos os elementos requeridos pelo critério, diz-se que tal conjunto *satisfaz* o critério.

O termo *Teste Baseado em Particionamento* (*Partition Testing*) é usado para se referir a um amplo conjunto de técnicas de teste que dividem o domínio de entrada do software em subdomínios (Hamlet e Taylor, 1990), (Weyuker e Jeng, 1991). O testador seleciona um ou mais dados de teste de cada um desses subdomínios. Na literatura de teste o termo “partição” é utilizado para se referir aos subdomínios resultantes desta divisão, sem o rigor matemático do termo, isto é, os subdomínios podem ter regiões sobrepostas ou podem ser disjuntos (partições no sentido matemático). Essas técnicas também são chamadas de *Baseadas em Subdomínio* (*Subdomain Testing*) por alguns autores (Ntafos, 1998). A premissa essencial de se realizar o particionamento do domínio é que os elementos de cada subdomínio sejam “similares” em algum sentido, e que sejam tratados

do mesmo modo pelo software. Caso esta premissa seja verdadeira, é suficiente testar apenas um elemento de cada subdomínio para avaliar o software.

Os critérios associados às técnicas de teste: Baseada em Especificação; Baseada em Estrutura; e Baseada em Defeitos podem, na sua maioria, ser classificados como técnicas baseadas em particionamento. Por exemplo, Os critérios Todos os Nós e Todos os Arcos dividem o domínio de entrada em subdomínios não disjuntos. Cada subdomínio consiste de todos os dados de teste que causam a execução do nó – ou arco. O critério Particionamento em Classes de Equivalência define subdomínios que agrupam dados de teste tratados do mesmo modo segundo a especificação do software.

O conceito de “técnicas baseadas em particionamento” pode ser aplicado para analisar também o Teste Exaustivo e o Teste Aleatório como formas extremas de partições. O Teste Exaustivo, conforme já definido, requer que cada elemento do domínio de entrada seja testado; portanto, o domínio de entrada é dividido em partições, cada uma consistindo de um único dado de teste – um elemento do domínio de entrada do programa. O Teste Aleatório pode ser visto como uma forma degenerada de particionamento em que há apenas uma partição, equivalente a todo o domínio de entrada (Weyuker e Jeng, 1991).

O Apêndice C descreve análises realizadas sobre as técnicas baseadas em particionamento. Essas análises identificam que a premissa de que dados pertencentes a um subdomínio sejam tratados da mesma forma pode não ser satisfeita em muitos casos, fazendo com que as técnicas que realizam o particionamento do domínio tenham eficácia próxima do Teste Aleatório (descrito na próxima seção).

A seguir são descritos alguns critérios de teste associados às técnicas de teste apresentadas nesta seção. Não se trata de uma lista exaustiva de critérios, buscou-se destacar critérios representativos de cada técnica para fornecer uma visão ampla das opções disponíveis para a seleção e avaliação de casos de teste. Tendo em vista o objetivo do capítulo em estabelecer uma base conceitual em técnicas e critérios sistemáticos de teste, não foram abordadas as *Técnicas Baseadas em Experiência* que, embora úteis à prática, são menos formais e não sistematizam a seleção e avaliação de conjuntos de dados de teste.

2.6.2. Técnicas e Critérios de Teste Baseados em Especificação e em Modelos

Técnicas de teste Baseadas em Especificação caracterizam-se por utilizar informações originadas da especificação do software como base para a seleção de casos de teste. A ideia essencial da técnica Baseada em Especificação é selecionar dados de teste representativos o suficiente para avaliar se as funcionalidades definidas na especificação do software foram implementadas corretamente ou não (Beizer, 1990), (Ostrand e Balcer, 1988), (SWEBOK, 2004). A natureza da informação do software utilizada no teste pode variar expressivamente, podendo ser, por exemplo: descrições sobre a interface do software; documentos de requisitos do software; ou modelos do software.

Teste Aleatório

No Teste Aleatório (TA), os dados de teste são selecionados aleatoriamente no domínio de entrada do software (Hamlet, 2005), (Ntafos, 1988), (Ntafos, 2001). Neste sentido, o TA pode ser considerado como um experimento de amostragem, cujos resultados podem ser submetidos a análises estatísticas e gerar estimativas de confiabilidade. A seleção aleatória é feita segundo alguma distribuição estatística, normalmente uma distribuição uniforme, isto é, cada possível dado de teste (um elemento do domínio de entrada) possui a mesma chance de ser selecionado. No entanto, o teste aleatório pode levar em conta alguma expectativa de como o software será usado (o perfil operacional) como base para uma seleção direcionada dos dados de teste. Neste caso, este teste é referenciado também como Teste Estatístico.

Um *perfil operacional*, ou *perfil de uso*, modela como se espera que o software seja utilizado em um contexto real de uso e pondera as entradas de acordo com o quão provável é que elas ocorram na prática (Crespo et al. 2008), (Musa et al., 1987). Este perfil é então utilizado como base para a geração de dados de teste, feita por meio de um gerador pseudo-aleatório. No nível de unidades o perfil operacional é definido pelo contexto da unidade, isto é, pelos valores passados por outras unidades identificadas na arquitetura do software.

No nível de sistema o perfil operacional é o perfil de uso, que descreve como o usuário (ou um outro sistema) irá interagir com o software. Em ambos os casos (nível de unidades de ou sistema) o perfil operacional pode ser visto como uma “quantificação refinada do domínio de entrada” (Hamlet e Taylor, 1990), (Hamlet, 2006).

No Teste Aleatório, quando os dados de teste são selecionados conforme o perfil operacional do software, os resultados do teste, em termos de falhas observadas, servem como uma estimativa do comportamento do software em uso real. Deste modo, a confiabilidade do software aferida durante o teste serve como uma estimativa da confiabilidade do software em uso.

O Teste Aleatório é frequentemente contraposto a técnicas baseadas em particionamento. Historicamente há controvérsias sobre a eficácia do TA e quando ele deve ser aplicado. Myers, por exemplo, descreveu a técnica como “provavelmente a mais pobre metodologia para o projeto de teste” (Myers, 1979). Este julgamento negativo origina-se provavelmente no fato de que o TA não utiliza informações mais elaboradas e específicas sobre o software em teste. De outro lado, trabalhos defendem o uso da técnica. Por exemplo, Thayer et. al. recomendam uma fase final de teste feita com dados selecionados aleatoriamente (Thayer et al., 1978); Mills et. al. incluem o teste aleatório usando o perfil operacional como parte do processo Cleanroom (Mills et al., 1987).

Análises empíricas e teóricas têm sido realizadas com resultados muitas vezes positivos para o TA. Duran e Ntafos apresentam resultados de simulações e um experimento que sugerem que o TA pode ter melhor custo-benefício que técnicas baseadas em particionamento (Duran e Ntafos, 1984), (Ntafos, 1998), (Ntafos, 2001). Isto porque as técnicas têm eficácia comparável e o TA oferece um custo menor para a geração dos casos de teste. Resultados similares foram obtidos por Hamlet e Taylor (Hamlet e Taylor, 1990), que também afirmam que o teste baseado em particionamento é superior ao teste aleatório (em termos de eficácia) apenas quando os subdomínios são associados a altas taxas de falhas, mas garantir que os subdomínios tenham esta característica é impossível. Isto requer um conhecimento *a priori* da localização dos defeitos, o que é, naturalmente, inviável.

Um aspecto problemático da técnica é que para que se ganhe confiança no software por meio do teste aleatório, é necessário um número expressivo de casos de teste (Thayer et al., 1978). Isto é, para que os resultados do teste sejam significativos é preciso que a amostra colhida aleatoriamente do domínio de entrada seja representativa. A geração aleatória dos valores de entrada tende a ser pouco complexa se comparada a outras técnicas, mas a avaliação das saídas produzidas tem custo alto e proporcional ao número de dados de teste gerados. Portanto, uma maior dificuldade para avaliar as saídas produzidas pelo software dificulta o uso do TA. A situação oposta, existência de um oráculo que permita checar as saídas produzidas mecanicamente, favorece a aplicação da técnica (Wichmann, 2011).

Trabalhos recentes incluem análises sobre o Teste Aleatório e aplicações da técnica em vários cenários. Alguns exemplos: Hamlet descreve uma análise de situações e tipos de software em que o TA é a melhor técnica, ou a única viável (Hamlet, 2006); Arcuri et al. realizam o estudo teórico de propriedades em termos de eficácia, previsibilidade, escalabilidade e limitações (Arcuri et al. 2010); Pacheco et al (2007) aplicam o TA para o teste de unidades de software orientado a objetos, no qual os testes (sequências de chamadas de métodos) são gerados iterativamente usando entradas anteriores já geradas para criar sequências válidas e novas de invocações e argumentos. É feita também a avaliação de componentes complexos da arquitetura Dot Net, com a revelação de defeitos não encontrados anteriormente (Forrester e Miller, 2000). Gotlieb et. al (2006) fazem a seleção aleatória de dados de teste para executar um conjunto de caminhos no software, representados por meio de condições geradas usando execução simbólica. Chen et al (2005) propuseram o Teste Aleatório Adaptativo que direciona a geração aleatória buscando revelar regiões de falha contínuas no domínio de entrada. Heurísticas são utilizadas para evitar a geração de dados de teste muito próximos uns dos outros. Esta técnica é descrita mais detalhadamente no Capítulo 4.

Particionamento em Classes de Equivalência

O critério *Particionamento de Equivalência* divide o domínio de entrada do software em classes de equivalência e requer que pelo menos um dado de teste de cada classe seja selecionado e executado (Beizer, 1990), (Pressman, 2006), (Sommerville, 2006). Essas classes são definidas pressupondo que todo dado de entrada pertencente a uma classe é tratado do mesmo modo, de acordo com a especificação do software. Portanto, cada dado de teste selecionado representa os demais dados de testes da sua classe de equivalência.

Por exemplo, se a especificação define que um software recebe como entrada uma variável *idade*, tipo inteiro, que pode variar de 0 a 120 anos (incluindo os extremos), podem ser identificadas as seguintes classes de equivalência:

Classe 1: idade menor que zero – referida como classe de valores inválidos;

Classe 2: idade entre 0 e 120 – classe de valores válidos; e,

Classe 3: idade maior que 120 – outra classe de valores inválidos.

Ao se executar o software com dados de teste que pertençam às Classes 1 e 3 espera-se que os valores fornecidos sejam identificados como não aceitáveis e sejam emitidas mensagens apropriadas. Ao se executar o software com um dado de teste que pertença à Classe 2 o valor fornecido deve ser aceito e uma saída apropriada deve ser produzida.

A Figura 2 ilustra as classes de equivalência do exemplo e os dados de teste selecionados para exercitar cada classe.

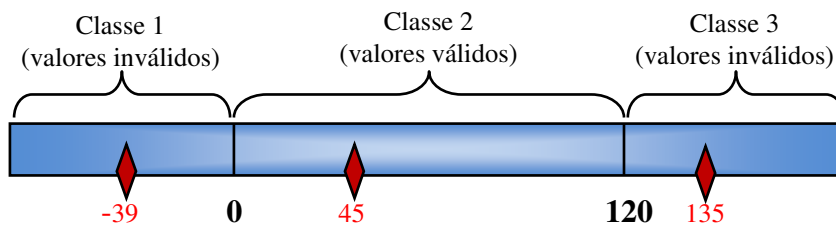


Figura 2. Classes de Equivalência e Dados de Teste

A definição de classes válidas e inválidas deve ser feita para todas as variáveis de entrada do software. Classes de equivalência podem também ser definidas a partir do

domínio de saída do software. Nestes casos devem ser identificadas restrições nos dados de entrada do software associadas a cada classe identificada no domínio de saída.

Deve-se notar que o modo como as classes de equivalência são definidas influencia fortemente a eficácia do critério. Esta definição deve ser feita de forma cuidadosa, analisando-se a natureza de cada variável de entrada do programa e a especificação do software em teste.

Após a análise da especificação do software e a especificação das classes de equivalência, dados de teste devem ser determinados de forma que cada classe de equivalência identificada seja exercitada. Nesta tarefa, um mesmo dado de teste pode exercitar mais de uma classe de equivalência. Notar também que as classes são tratadas de forma independente uma das outras, isto é, não é requerido por este critério que combinações de classes de equivalência sejam exercitadas.

Análise de Valores Limites

O critério Análise de Valores Limite complementa o critério Particionamento de Equivalência (Beizer, 1990), (Pressman, 2006), (Sommerville, 2006). Frequentemente as situações de transição de comportamento do software de uma classe para outra classe, as situações limite, são implementadas de forma incorreta. Estas situações costumam ser inerentemente complexas e acabam induzindo a erros de análise, projeto ou codificação.

Para descobrir defeitos associados a estas situações é recomendada a seleção de dados de teste situados nos limites das classes. No exemplo da seção anterior (variável idade, tipo inteiro, que pode variar de 0 a 120 anos, incluindo os extremos) os valores limite seriam: 1, 0 e -1, referentes ao limite entre as Classes 1 e 2; e 119, 120, 121, referentes ao limite entre as Classes 2 e 3, conforme ilustrado na Figura 3. Notar que são selecionados valores exatamente no limite, valores imediatamente inferiores e imediatamente superiores ao limite.

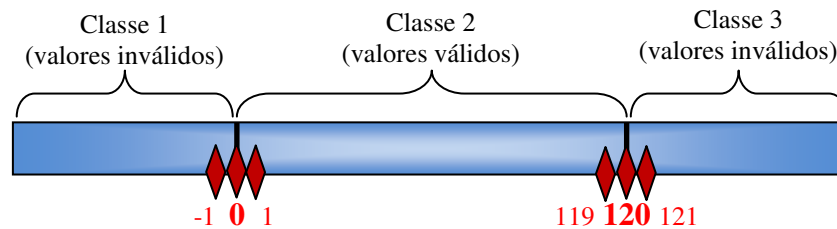


Figura 3. Valores Limite e Dados de Teste

Grafo Causa-Efeito

O critério Grafo Causa-efeito permite a seleção de dados de teste de um modelo que representa combinações de condições de entrada e as respectivas ações a serem produzidas como saída (Beizer, 1990), (Binder, 2000), (Pressman, 2006), (Sommerville, 2006). Este grafo pode ser visto como uma ferramenta para identificar e representar relacionamentos entre entradas e saídas e selecionar casos de teste. Tipicamente esses casos de teste podem ser representados por meio de uma tabela de decisão.

No Grafo Causa-efeito um nó é definido para cada causa (entrada) e para cada efeito (saída ou resultado). Os nós associados a causas e os associados a efeitos ficam em lados opostos. Uma linha ligando a causa ao efeito indica que a causa é uma condição necessária para o efeito. Se um efeito está associado (por uma linha) a apenas uma causa, esta causa é uma condição suficiente para que o efeito ocorra. Se um efeito é associado a duas ou mais causas, o relacionamento lógico entre as causas é descrito por símbolos indicando uma conexão lógica *e* (^); uma conexão lógica *ou* (v); uma causa cuja negação é uma condição necessária ao efeito é indicada com uma conexão lógica *não* (~).

Dados de teste são derivados pela realização dos seguintes passos: são identificadas as causas e os efeitos pela análise da especificação do software. As causas correspondem a entradas para o software em teste, enquanto que os efeitos são qualquer saída a ser produzida. É feita então a análise da semântica da especificação para gerar um grafo Causa-efeito, ligando as entradas às saídas e determinando os relacionamentos lógicos existentes; são identificadas combinações entre causas e efeitos que são impossíveis devido a

restrições do problema. Por fim o grafo é convertido em uma tabela de decisão, na qual cada coluna representa um caso de teste.

Tabelas de Decisão

Tabelas de Decisão são uma importante ferramenta para o teste baseado em combinações. O uso dessas tabelas no teste de software é inspirado em fundamentação conceitual do teste de hardware e tem primeiras aplicações desenvolvidas por volta de 1970 (Goodenough e Gerhart, 1975).

Binder (2000) destaca que o uso de tabelas de decisão é indicado quando o software possui as seguintes características: 1) respostas distintas são selecionadas de acordo com situações distintas das variáveis de entrada; 2) essas situações podem ser descritas como expressões lógicas mutuamente exclusivas em função das variáveis de entrada; 3) a resposta a ser produzida não depende da ordem em que as variáveis de entrada são fornecidas; e 4) a resposta a ser produzida não depende de entradas ou saídas anteriores. Se estas características forem observadas, o uso de tabelas de decisão pode ser útil.

Tabelas de Decisão são formadas por duas partes: a *seção de condições* e a *seção de ações*. A *seção de condições* lista condições e combinações de condições. *Variáveis de decisão* são as entradas referenciadas nesta seção, uma *condição* expressa um relacionamento entre *variáveis de decisão* e deve, ao ser resolvida, assumir o valor *verdadeiro* ou o valor *falso*. Cada combinação única de condições e ações respectivas é chamada de uma *variação* e é representada tipicamente em uma única coluna (ou linha) da tabela.

A *seção de ações* lista as *respostas* a serem produzidas quando as respectivas combinações de condições forem verdadeiras, conforme determinado pelos valores das variáveis de decisão.

Outros pontos que merecem menção: cada ação deve ser produzida por pelo menos uma variação; se mais de uma combinação de condições resulta na mesma ação, variações explícitas para combinação devem ser criadas; podem existir valores do tipo “não importa”

para designar condições que, para uma variação particular, podem ser verdadeiras ou falsas sem alterar a ação resultante; podem existir valores do tipo “não pode ocorrer” para designar condições que sejam mutuamente excludentes.

Teste Combinatório

As técnicas de teste baseadas em combinação apresentadas anteriormente enfatizam o uso de ferramentas (grafos ou tabelas) para representar ou organizar possíveis combinações de entradas e respectivas saídas. Uma abordagem que tem sido desenvolvida mais recentemente destaca a redução do número de casos de teste a serem aplicados, em relação a todas as combinações de entradas, por meio da utilização de conjuntos de casos de teste que exercitem tipos específicos de combinações de entradas.

Esta abordagem pode ser considerada como uma alternativa entre os extremos: o teste baseado em classes de equivalência (não requer a combinação de valores) e o teste exaustivo (requer todas as combinações). Conforme Dalal et al (1999) um “teste default” pode ser feito selecionando um valor default para cada parâmetro e variando um parâmetro em cada teste, até que todos os valores para todos os parâmetros tenham sido testados pelo menos uma vez. Ou seja, neste teste, cada possível valor para cada parâmetro é exercitado. Esta estratégia é também chamada de Um Fator Por Vez (*One Factor One Time*), ou de Estratégia de Combinação de Escolha Base (*Base Choice Combination Strategy*). Abordagens para o teste combinatório requerem que combinações de valores para os parâmetros sejam exercitadas, estendendo de certa forma o teste com a estratégia Um Fator Por Vez.

O teste combinatório, nesta perspectiva mais recente, é baseado na premissa de que muitas falhas em software acontecem por meio da interação de alguns parâmetros de entrada; nesses casos, dois ou mais valores de entrada interagem fazendo com que o software produza um resultado incorreto. Dito de outra forma: nem todo parâmetro contribui para toda falha, e a maioria das falhas é causada pela interação entre relativamente poucos parâmetros (Bryce et al., 2010), (Kuhn et al., 2004), (Kuhn et al., 2009).

Evidências empíricas deste fato são apresentadas por Kuhn et al (2004). Foram analisados relatórios de falhas em software de diferentes naturezas: embarcado, sistema operacional, servidor HTTP, browser web e científico. Apesar de variações conforme a natureza do software, como comportamento geral observou-se que o número de parâmetros que contribui para a maioria das falhas é de um, dois, ou três. São raras as situações em que valores específicos para cinco ou seis parâmetros são requeridos para provocar a falha.

Baseado na premissa e evidência anteriores abordagens têm sido propostas de testar combinações específicas de parâmetros usando modelos combinatórios como Vetores Ortogonais, Vetores de Cobertura e outras técnicas (Cohen D. et al., 1997), (Cohen M. et al., 2003), (Dalal et al., 1999), (Grindal et al., 2005), (Hoskins et al., 2004), (Mohammad et al., 2008), (Nie e Leung, 2009). Essencialmente, ao invés de combinar todos possíveis valores, de todos os possíveis parâmetros, são construídas e testadas combinações específicas, caracterizadas pelo nível de interação entre parâmetros, referido como n . Para $n = 2$ tem-se o teste de todos os possíveis valores de pares de parâmetros de entrada, chamada de *interação 2-modos* (2-way ou *pairwise*). Com $n = 3$ é feito o teste de combinações valores de triplas de parâmetros (3-modos). Genericamente, o teste n -modos significa o teste de combinações de parâmetros de ordem n .

O Apêndice A apresenta mais informações sobre abordagens de teste combinatório: definições e notação importantes, principais características e modelos, e o processo do teste combinatório.

Teste Baseado em Modelos

O Teste de Software Baseado em Modelos (*Model-Based Software Testing*) vem ganhando espaço nos últimos anos devido ao avanço do paradigma de orientação a objetos e à evolução da adoção de modelos na engenharia de software – o desenvolvimento de software baseado em modelos (Stahl e Völter, 2006).

El-Far e Whittaker (2001) definem um modelo como “uma descrição do comportamento de um software”. Bertolino et al (2005) definem o Teste Baseado em

Modelos (TBM) como “a derivação de casos de teste de um modelo representado o comportamento do software”. Dalal et al (1999) enfatizam o uso de um modelo de dados que especifica as entradas recebidas pelo software (interface do software). O teste baseado em modelos refere-se, portanto, a abordagens que baseiam as tarefas de teste, como a geração de dados de teste e a avaliação de resultados, em um modelo do software em teste.

Uma taxonomia para o teste baseado em modelos é proposta em Utting e Legeard, (2006) e Utting et al (2006). São identificadas sete diferentes dimensões para analisar estas técnicas e são discutidas possíveis instanciações em cada dimensão. Essas dimensões, embora ortogonais, influenciam-se mutuamente e relacionam-se essencialmente aos aspectos: características dos modelos utilizados; critério de seleção de dados de teste; técnicas para automação da geração de dados e para a execução do teste.

Em uma revisão sistemática as abordagens de TBM (relatadas em 202 artigos) foram classificadas nas seguintes categorias (Dias Neto et al., 2007): i) modelos que representam informações sobre requisitos e usam diagramas UML; ii) modelos que representam informações sobre requisitos e usam qualquer notação não UML; iii) modelos que representam informações da estrutura do software (arquitetura, componentes, interfaces) descrita usando diagramas UML; e iv) modelos que representam informações da estrutura do software usando qualquer notação não UML. Esta revisão apresenta informações que caracterizam os trabalhos em TBM em relação a aspectos como: nível do teste, tipos de modelos e custo de utilização.

Shafique e Labiche (2010) apresentam uma revisão sistemática sobre ferramentas de suporte para o TBM. O trabalho tem como escopo ferramentas “baseadas em estados” (*state-based model based testing tools*), tanto comerciais quanto voltadas à pesquisa. A questão investigada é quais são as ferramentas de TBM e quais são as suas características. É identificado o nível de suporte automatizado para atividades como: criação do modelo; verificação do modelo; depuração; rastreabilidade de requisitos; e criação de *drivers*, *stubs* e oráculos, geração de casos de teste, e avaliação de adequação de casos de teste.

O Apêndice A apresenta mais informações sobre abordagens de teste baseado em Modelos. Além de detalhar os trabalhos citados nesta seção, são abordados os seguintes tópicos e trabalhos:

- Tarefas fundamentais para a realização de um teste baseado em modelos (El-Far e Whittaker, 2001), (Dias Neto et. al, 2007), (Binder, 2000), (El-Far e Whittaker, 2001), (Utting e Legeard, 2006), (Walton e Poore, 2000).
- Um histórico de aplicações e soluções de teste baseado em modelos, incluindo a descrição dos trabalhos: (Clarke, 1988), (Apfelbaum e Doyle, 1997), (Offutt e Abdurazik, 1999), (Fröhlich e Link, 2000), (Fantinato e Jino, 2003), (Kansomkeat e Rivepiboon, 2003), (Kansomkeat e Rivepiboon, 2003), (Hartman e Nagin, 2004), (Bertolino e Marchetti, 2005), (Bringmann e Krämer, 2006), (Hemmati et al., 2010), (Sarma et al., 2010).
- Abordagens de Teste Baseado em Casos de Uso: (Binder 2000), (Briand e Labiche, 2001), (Carnielo, Chaim e Jino, 2004).
- Teste Baseado em Máquinas de Estados Finitos: (Chow, 1978), (Simão, 2007).

2.6.3. Técnica de Teste Baseada em Estrutura

Na técnica de Teste Baseada em Estrutura (também referida como técnica de teste estrutural) os dados de teste são selecionados por meio da análise da estrutura interna do software. Portanto, o código fonte do software deve estar disponível, pois ele é utilizado para a seleção dos dados de teste. A motivação para o uso desta técnica muitas vezes é feita com questões do tipo: “você confiaria em um software se fosse informado que algumas instruções deste software nunca foram executadas?” Esta pergunta enfatiza que defeitos em trechos não exercitados do código fonte do software certamente não serão descobertos (Myers, 1979).

Podem ser distinguidas duas abordagens para o teste baseado em estrutura: uma voltada aos aspectos de fluxo de controle do software e outra voltada aos aspectos de fluxo de dados do software. Para uma descrição detalhada, veja (Barbosa et al., 2007).

Conceitos Iniciais

Na Técnica Baseada em Estrutura uma unidade de software (função, procedimento ou método) é considerada como o conjunto de seus componentes estruturais. Estes componentes estruturais podem ser comandos de desvio (como os comandos: *if-then*; *switch*; *while* e *repeat*) ou outros comandos (instruções como: $x = y + 2$; $read(a,b)$; $x = func(a,b)$). Ao serem executados, os comandos de desvio fazem a seleção da computação a ser realizada a seguir, baseada na avaliação do predicado associado ao comando.

O software pode ser representado por um *Grafo de Fluxo de Controle* (GFC) composto de *nós* – correspondem a blocos de comandos que são sempre executados conjuntamente –, e *ramos*, que correspondem a possíveis desvios no fluxo de execução, determinados pelos comandos de desvio. O GFC possui um único nó de entrada, um único nó de saída, e define um conjunto de *caminhos* que podem ser executados do nó de entrada ao nó de saída, passando por certos comandos e desvios ao longo da execução. Quando se executa o software com um dado de teste, algum caminho específico é executado no programa. Isto só não ocorre caso o software entre em um *laço infinito*, situação na qual um sub-caminho infinito é executado.

Durante a execução do software valores são atribuídos a variáveis (conferidos a uma posição na memória associada à variável) e são utilizados (recuperados da posição na memória associada à variável). Por exemplo, a execução do comando $x = y + 2$ faz com que o valor de y seja recuperado (uso de y), seja somado a 2, e atribuído à variável x (*definição* de x). *Modelos de fluxo de dados* determinam quando uma ocorrência de uma variável no software é uma *definição* da variável e quando uma ocorrência é um *uso* da variável. Quando se executa o software com um dado de teste, variáveis são definidas e usadas, o que caracteriza a existência de um *fluxo de dados* na execução.

A Figura 4 mostra um código fonte em C (*identifier.c*) e o respectivo GFC (Barbosa et al., 2000). Notar que o código fonte apresenta no lado esquerdo como comentários o número do nó do GFC ao qual o comando pertence. É possível identificar conjuntos de comandos executados sempre conjuntamente, por exemplo, os comandos precedidos por /*

01 */ que pertencem ao nó 1 do GFC. Pode-se também observar o efeito de comandos de desvio no fluxo de controle, por exemplo, o comando while presente no nó 4 do grafo, ou o comando if-then-else iniciado no nó 8.

A variável *length* possui definições nos nós: 1, 2 e 7 e possui usos nos nós 7 e 8. Observar que no nó 7 a variável é incrementada (*length++*) o que caracteriza um uso seguido de uma definição da mesma variável. No nó 8 há dois usos da variável, ambos em condições de um predicado (*length ≥ 1 e length < 6*).

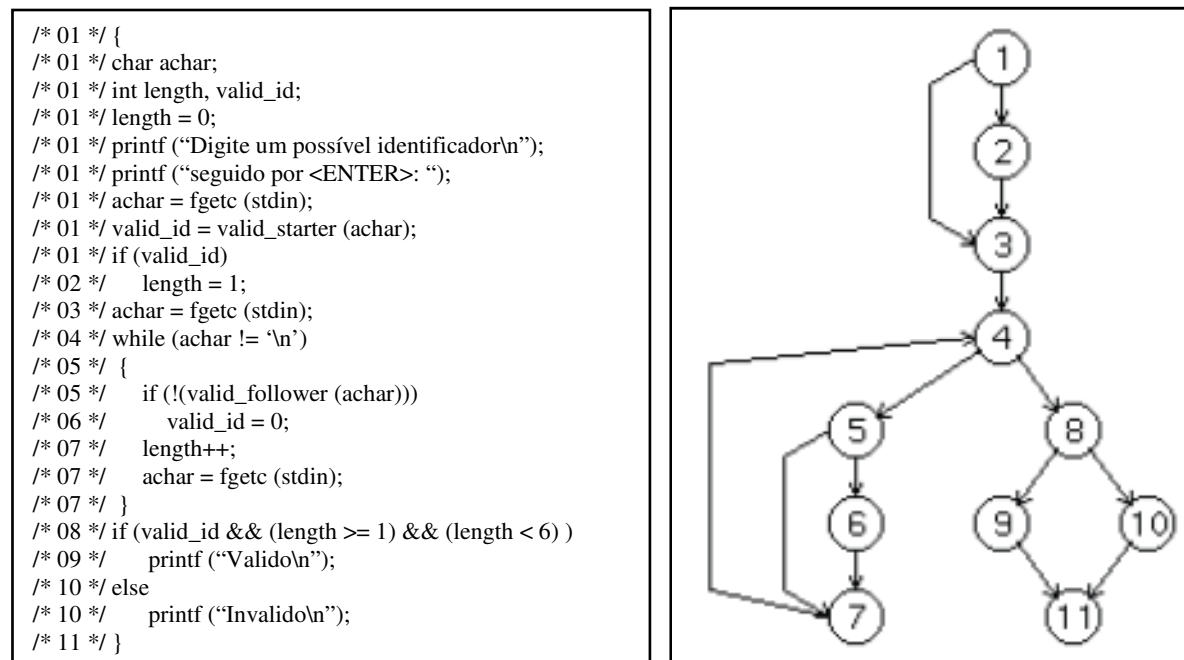


Figura 4. Código Fonte e Respetivo Grafo de Fluxo de Controle

Crerários de teste Baseados em Estrutura estabelecem componentes internos do software como elementos requeridos. Esses componentes internos podem ter natureza de fluxo de controle ou de fluxo de dados, levando a uma classificação dessas técnicas:

- Técnicas Baseada em Estrutura com ênfase em fluxo de controle (baseadas em fluxo de controle) utilizam informações de fluxo de controle do programa como base para a seleção de dados de teste, de tal forma que determinados tipos de estrutura do grafo do programa sejam exercitadas.

- Técnicas Baseada em Estrutura com ênfase em fluxo de dados (baseadas em fluxo de dados) utilizam informações do fluxo dos dados existente no programa, visando identificar atribuições e utilizações das variáveis através do programa, gerando componentes elementares a serem exercitados pelo testador.

CrITÉRIOS de teste Baseados em Estrutura são aplicados principalmente no nível de unidade, mas também no nível de integração. Em geral técnicas e critérios são desenvolvidos inicialmente focando o teste de unidades e são adaptados, em um segundo momento, para tratar também o teste de integração das unidades.

O teste de integração concentra-se nas relações e interfaces entre os módulos. Ou seja, enquanto no teste de módulo consideram-se separadamente cada parte distinta do programa (procedimento, função, ou método), testando os algoritmos e estruturas de dados; no teste de integração consideram-se várias partes e testam-se as relações e interfaces entre elas. As relações entre os módulos são representadas pelas chamadas de procedimento, função (ou método) presentes nos módulos e podem ser representadas por meio de um Grafo de Chamadas (Ryder, 1979). As interfaces entre os módulos possibilitam a comunicação entre eles, seja por meio de passagem de parâmetros, seja através de variáveis de escopo global.

Neste capítulo restringimo-nos às técnicas e critérios para teste de unidades. Técnicas e critérios estruturais de integração são descritos em Harrold e Soffa (1991), Linnenkugel e Mülerburg (1990), Pande et al (1994) e Vilela, (1998).

Uma linha de trabalho importante consiste na análise e comparação de critérios de teste baseados em estrutura por meio de estudos teóricos e empíricos. Não é objetivo detalhar tais estudos, apenas são descritas suas naturezas e objetivos:

- Estudos teóricos buscam, em geral, avaliar a relação entre os critérios em termos de inclusão determinar a complexidade dos critérios, exemplos: (Clarke et al., 1989), (Demillo et al.,1995), (Frankl e Weyuker, 1993-a), (Frankl e Weyuker, 1993-b), (Howden, 1976), (Ntafos, 1988), (Weyuker e Jeng, 1991). A análise de inclusão estabelece uma hierarquia entre os critérios de teste. Um critério *CI* inclui o critério

$C2$ se, para qualquer programa P , qualquer conjunto de dados de teste D que satisfaz $C1$ também satisfaz $C2$. A análise de complexidade provê uma estimativa do custo da aplicação do critério, considerando o número total de dados de teste requerido para satisfazer um critério, no pior caso.

- Estudos empíricos buscam, em geral, avaliar o custo real da aplicação de critérios e a eficácia dos critérios para revelar defeitos existentes, exemplos: (Bieman e Schultz, 1992), (Frankl e Weiss, 1993), (Weyuker, 1990), (Weyuker, 1993). Em relação ao custo, trabalhos abordam modelos para prever o número de dados de teste necessários para satisfazer os critérios; e avaliam o número elementos requeridos não executáveis em relação ao total de elementos requeridos pelos critérios. Quanto à eficácia estudos avaliam a capacidade de revelar defeitos dos critérios e realizam comparações entre os critérios em relação a esta capacidade.

Introdução à Técnica Baseada em Fluxo de Controle

Os primeiros critérios de teste estrutural eram baseados essencialmente no fluxo de controle do software. Em Myers pode ser encontrada uma descrição detalhada desses critérios (Myers, 1979). Zhu et al (1993) apresentam um sumário bastante completo sobre técnicas de teste voltadas para o teste de unidades.

Os principais critérios desta técnica são descritos a seguir: *Todos os Comandos*, *Todos os Ramos*, *Todos os Caminhos* e Cobertura de Condições Múltiplas.

O critério de teste *Todos os Comandos* (ou *Todos os Nós*) requer que cada comando do software seja exercitado pelo menos uma vez durante o teste. A determinação de dados de teste que exercitem um comando específico requer a análise do código fonte do software e a escolha de valores para as variáveis de entrada que provoquem a execução do comando.

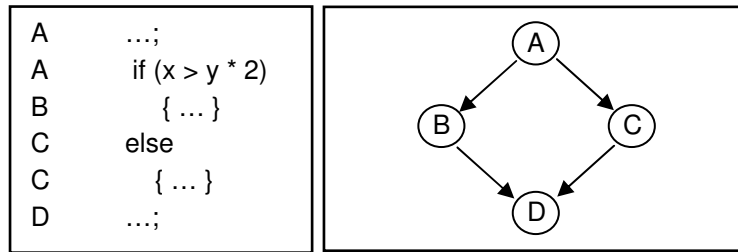


Figura 5. Comando if-then-else e Respetivo Grafo de Fluxo de Controle

Por exemplo, a Figura 5 mostra um trecho de programa no qual há um comando de decisão *if-then-else* e o respectivo GFC. Para que os comandos representados por B sejam executados é necessária a seleção de valores de entrada tais que a *condição lógica* do comando *if* seja satisfeita; por exemplo, valores de entrada que façam com que x tenha valor 100 e y tenha valor 30 imediatamente antes do comando *if* (notar que x e y podem ser variáveis de entrada, mas também podem ser computadas internamente, a partir de outras variáveis). Outros dados de teste serão necessários para exercitar os comandos representados por C; por exemplo, dados que resultem nos valores $x = 20$ e $y = 15$ antes do comando *if*.

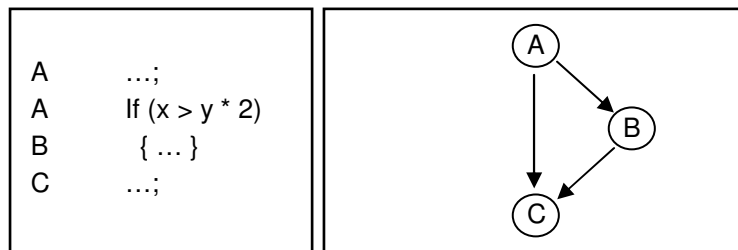


Figura 6. Comando if-then e Respetivo Grafo de Fluxo de Controle

O critério *Todos os Ramos* requer que cada possível transferência de controle seja exercitada pelo menos uma vez. A Figura 6 mostra um trecho de programa com um comando condicional *if-then*. Os dados de teste que satisfazem a condição do *if*, exercitam os comandos representados em B e também os em C. Este critério, no entanto (ao contrário do anterior), requer a execução de dados de teste adicionais, que provoquem a transferência de controle (ou ramo) do comando *if* diretamente para os comandos em C (sem passar por B).

O critério *Todos os Caminhos* requer que cada diferente caminho no GFC seja executado pelo menos uma vez. Este critério é geralmente considerado excessivamente exigente visto que, em programas reais, a sua aplicação tende a gerar conjuntos muito grandes de elementos requeridos. Considere a Figura 7 que mostra um trecho de código com dois comandos *if-then-else* em sequência e o respectivo GFC; note que o segundo comando *if* pertence ao nó D no grafo. A execução de apenas dois casos de teste é suficiente para exercitar todos os nós e também todos os arcos do GFC; por exemplo, utilizando um caso de teste que exercite a sequência ABDEG e outro que exercite a sequência ACDFG. No entanto, para satisfazer o critério todos os caminhos são necessários quatro casos de teste, cada um exercitando um caminho do GFC: ABDEG, ABDFG, ACDEG e ACDFG.

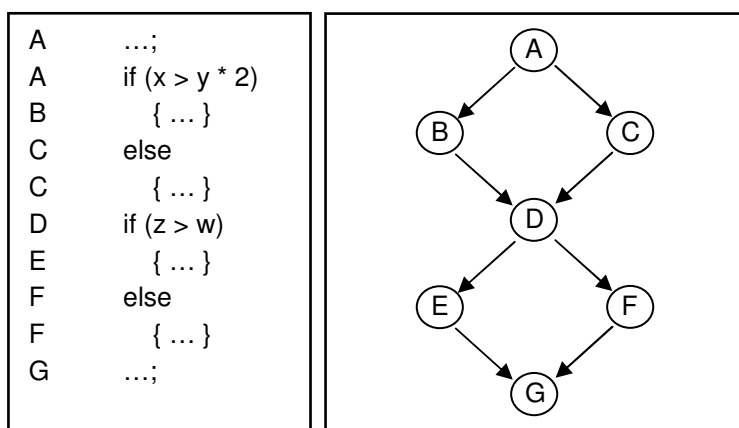


Figura 7. Comandos if-then em Sequência e Respectivo Grafo de Fluxo de Controle

O teste utilizando o critério *Todos os Caminhos* é especialmente complexo em caso de programas que possuem laços. A cada nova iteração de um laço tem-se um novo caminho, o que leva a um número potencialmente infinito de caminhos.

Outra abordagem para tratar o teste de caminhos de um software é utilizar informações sobre a complexidade do software e selecionar caminhos para o teste. O critério de McCabe, ou *Teste de Caminhos Básicos* é um exemplo (McCabe, 1976). A Complexidade Ciclomática, computada com base no grafo de fluxo de controle do software, oferece um limite máximo para o número de caminhos linearmente independentes no grafo. Um caminho linearmente independente é qualquer caminho que introduza pelo

menos um novo conjunto de instruções ou uma nova condição. Ao se selecionar caminhos independentes e executá-los, garante-se que todos os comandos e todas as condições foram exercitados.

O critério *Cobertura de Condições Múltiplas* direciona o testador a selecionar dados de teste de tal forma que todas as condições no software sejam exercitadas, e ainda, que cada condição de uma *condição composta* seja também exercitada pelo menos uma vez (Myers, 1979). Uma *condição composta* é uma condição na qual duas ou mais condições simples são unidas por meio de operadores lógicos.

Isto significa que se o software em teste possuir uma condição composta, todas possíveis combinações de avaliações (verdadeiro ou falso) das condições que formam a condição composta devem ser exercitadas. Considerando uma condição composta CC formada por duas condições simples C_1 e C_2 ligadas por um operador lógico OL CC : $(C_1 OL C_2)$ o critério determina que C_1 e C_2 (C_1, C_2) sejam avaliadas como: (verdadeiro, verdadeiro), (verdadeiro, falso), (falso, verdadeiro) e (falso, falso).

O Teste de Predicados, abordado por Tai considera o tratamento de predicados compostos (Tai, 1996). As técnicas propostas buscam caracterizar defeitos em predicados, tais como defeitos em operadores lógicos e defeitos em operadores relacionais. Dados de teste são selecionados visando garantir a detecção desses defeitos. Tais técnicas são normalmente definidas como Baseadas em Defeitos (descritas em seção posterior).

Introdução à Técnica Baseada em Fluxo de Dados

O processamento de entradas fornecidas ao software e a produção de resultados ocorre essencialmente pela manipulação de dados. Nessa perspectiva abstrata o software (S) pode ser considerado como uma função de um domínio de entrada (DE) para um domínio de saída (DS), $S: DE \rightarrow DS$. Para uma entrada o software produz uma saída correspondente por meio de uma computação d_1, d_2, \dots, d_n , que é uma sequência de estados de dados começando com um estado de entrada d_1 , e terminando com um estado de saída d_n . Este modelo permite considerar o software como uma composição de mapeamentos s_1, s_2, \dots, s_n .

s_i , na qual $s_i: d_i \rightarrow d_{i+1}$. (Alshraideh et al., 2010). Esta abstração explicita que o resultado final produzido pelo software depende de uma sequencia de estados intermediários e de transformações realizadas entre esses estados. Esta cadeia de transformações baseia-se fundamentalmente no fluxo de dados que ocorre quando o software é executado.

A técnica estrutural baseada em fluxo de dados utiliza informações do fluxo dos dados existente no software, visando identificar atribuições e utilizações das variáveis no software e gerar componentes elementares a serem exercitados no teste (Frankl e Weyuker, 1988), (Herman, 1976), (Laski e Korel, 1983), (Maldonado, 1991), (Ntafos, 1984), (Rapps e Weyuker, 1985). Essencialmente, tais critérios exigem a execução de caminhos do ponto onde uma variável foi definida, ate o ponto onde ela foi utilizada (ou potencialmente utilizada (Maldonado, 1991)).

Por realizarem a seleção de caminhos no software para teste, estes critérios estabelecem uma ponte que preenche a lacuna existente entre critérios baseados em fluxo de controle menos exigentes (critérios Todos os Nós e Todos os Ramos) e o critério Todos os Caminhos, de aplicação inviável na maioria dos casos.

A ideia intuitiva para se levar em conta os fluxos de dados no teste é que não se deve considerar o software suficientemente testado se, para os pontos onde valores são atribuídos a variáveis, tais valores não tenham sido usados posteriormente. Se um valor atribuído durante a execução foi indevido, isto poderá ser descoberto quando tal valor for usado depois (Rapps e Weyuker, 1985).

A Figura 8 ilustra de forma simples uma aplicação da técnica baseada em fluxo de dados. No *nó 1* há uma definição da variável y . Há dois usos dessa variável, um uso no *nó 5* e outro uso no *nó 6*. Uma análise de fluxo de dados indica duas *associações definição-uso* (*def-uso*). Uma do *nó 1* ao *nó 5* com relação à variável y (descrita como $\langle 1,5,y \rangle$) e outra do *nó 1* ao *nó 6* com relação à variável y ($\langle 1,6,y \rangle$). Durante o teste, para que cada associação seja exercitada é necessário provocar a execução de um caminho que passe na definição e alcance o uso da variável. Por exemplo, a associação $\langle 1,5,y \rangle$ é exercitada quando o caminho $1,2,4,5,7$ é executado e também quando o caminho $1,3,4,5,7$ é executado.

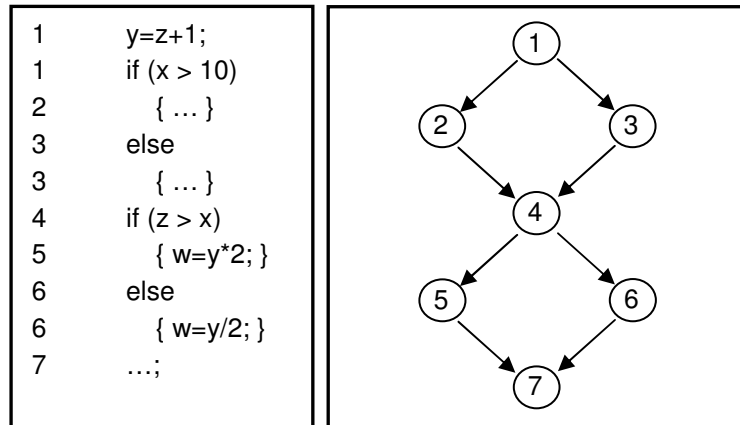


Figura 8. exemplo de aplicação da técnica baseada em análise de fluxo de dados

Herman definiu um dos primeiros critérios que usam informações do fluxo de dados (Herman, 1976). Tais critérios requerem que todo uso de uma variável seja exercitado pelo menos uma vez, a partir dos pontos no software onde a variável é definida. Tem-se, portanto o conceito de exercitar associações entre definições e usos de variáveis.

Rapps e Weyuker (1985) definiram uma família de critérios baseados em análise de fluxo de dados. Esses critérios requerem basicamente que dados de teste executem caminhos livres de definição de cada nó contendo uma definição global de uma variável, para nós contendo usos computacionais globais, e arcos contendo usos predicativos desta variável. Em relação ao critério definido por Herman, os critérios de Rapps e Weyuker introduzem a distinção de usos de variáveis como usos computacionais (c-usos) e usos predicativos (p-usos). Os principais critérios da família Rapps e Weyuker são descritos na próxima seção.

Maldonado et al (1991) propuseram uma família de critérios chamada de Potenciais Usos (Maldonado, 1991), (Maldonado, Chaim e Jino, 1992). A ideia básica desta família de critérios consiste em requerer associações independentemente da ocorrência explícita de uma referência a uma determinada definição de variável: se o uso desta definição pode existir, a potencial associação é requerida. Em outras palavras, os critérios requerem basicamente que caminhos livres de definição, em relação a qualquer nó *i*, que possua definição de variável e a qualquer variável *x* definida em *i*, sejam executados, independentemente de ocorrer uso dessa variável nesses caminhos.

Ao requererem-se potenciais usos, ao invés de usos, abre-se a possibilidade do critério levar o testador a selecionar dados de teste que possuam maior chance de revelar defeitos do tipo usos ausentes, ou seja, um uso que deveria existir e foi ignorado pelo programador. Outro aspecto importante desses critérios é que eles incluem o critério estrutural Todos os Ramos (Maldonado, 1991). Isto é, exercitar todos os elementos requeridos dos critérios potenciais usos garante que todos os ramos do programa serão exercitados. Quando se considera a existência de caminhos não executáveis os critérios Rapps e Weyuker não garantem esta propriedade.

Ntafos (1984) definiu um critério que requer que sequencias de interações definição uso sejam exercitadas. O elemento requerido é referido como uma *interação K-dr*. Uma interação *K-dr* consiste de $k-1$ variáveis x_1, x_2, \dots, x_{k-1} (não necessariamente distintas) e K comandos distintos s_1, s_2, \dots, s_k ; tal que existe um caminho no programa que visita os comandos na ordem dada, e a variável x_i é definida em s_i , sendo que esta definição atinge uma referência a x_i em s_{i+1} , a qual é usada para definir a variável x_{i+1} em s_{i+1} . Com i variando de 1 a K . A interação *k-dr* é essencialmente uma sequencia de definições e usos de variáveis, que estabelece uma cadeia de interferências, em que a variável x_i interage como a variável x_{i+1} , esta como a x_{i+2} , e assim por diante.

O critério de Ntafos é chamado de *k-tuplas requeridas*. O parâmetro K define o comprimento da interação *k-dr* requerida. Valores superiores de K resultam em critérios mais exigentes, segundo o autor; fazendo com que interações de fluxo de dados mais complexas sejam requeridas.

Laski e Korel (1983) definiram os critérios de teste: contexto elementar de dados e contexto ordenado de dados. Um Contexto Elementar de Dados de um comando s_i é definido como um subconjunto de variáveis $EDC(i)$ usadas no comando s_i , tal que existe um caminho do início do programa até s_i que incluía, para cada variável v de $EDC(i)$, caminhos livres de definição c.r.a v do ponto em que v é definida até s_i .

Deve-se perceber que o conceito de Contexto Elementar de Dados não envolve a consideração da ordem na qual as definições de variáveis aparecem. Tem-se outro conceito, o de Contexto Elementar Ordenado de Dados, que considera a ordem das definições. Neste

caso, é requerido que sequencias ordenadas de definições das variáveis de $EDC(i)$ sejam exercitadas.

No Apêndice A são apresentados, de maneira formal conceitos relativos à análise de fluxo de controle e de fluxo de dados e são definidos os critérios da técnica de teste baseada em estrutura. Neste apêndice é também descrita a Ferramenta Poke-Tool, utilizada na avaliação empírica descrita no Capítulo 5. São abordados os trabalhos: (Myers, 1979), (Rapps e Weyuker, 1985), (Chaim, 1991), (Maldonado, 1991), (Vergilio, 1992), (Poston e Sexton, 1992), (Zhu e Hall, 1993), (Chaim et al., 1995), (Yang et. al. 2006), (Barbosa et al., 2007) e (OSSTT, 2011).

2.6.4. Técnicas de Teste Baseadas em Defeitos

De acordo com Morell (1990) um teste é baseado em defeitos quando ele busca demonstrar que certos defeitos não estão presentes em um software. Nesta perspectiva, a execução de um teste que não provoca falhas, indica que o software não possui tipos específicos de defeitos. A informação obtida pela execução de casos de teste sem provocar falhas é a identificação de um conjunto de defeitos inexistentes no software.

A Análise de Mutantes é uma técnica baseada em defeitos amplamente desenvolvida, com suporte significativo conceitual e empírico (Delamaro e Maldonado, 1996), (DeMillo, Zipton e Sayward, 1978), (Fabbri et al., 1999), (Howden, 1982), (Jia e Harman, 2010). Além desta técnica, descrita na Seção seguinte, destacam-se as técnicas Teste de Domínios (Domain Testing) (Clarke et al., 1982), (White e Cohen, 1980). Teste Estrutural Baseado em Restrições (Vergilio, 1997).

O Teste de Domínios foi originalmente desenvolvido por White e Cohein (1980) para ser utilizado conjuntamente com o teste estrutural baseado em caminhos. A ideia é detectar defeitos de domínio, exercitando os limites do domínio do caminho e pontos próximos deste limite. O domínio de um caminho é definido como o conjunto de valores de entrada que provoca a sua execução. Os limites deste domínio são definidos pelas condições associadas aos ramos existentes ao longo do caminho. Esta técnica foi estendida em

trabalhos subsequentes, ex: (Clarke et al., 1982) e baseia-se em princípios semelhantes à técnica Análise de Valores Limites, já descrita.

Critérios de teste estrutural baseados em restrições foram definidos por Vergilio (1997). Tais critérios combinam elementos requeridos por critérios estruturais, como ramos, usos e caminhos com restrições que descrevem defeitos de diferentes tipos. Estas restrições de cobertura representam condições necessárias para que os componentes estruturais executados sejam considerados efetivamente exercitados.

Análise de Mutantes

O Teste Baseado em Análise de Mutantes, ou Teste de Mutantes (*Mutation Testing*) é uma técnica de teste baseada em defeitos, que tem como ideia básica a de que defeitos considerados no teste representam enganos que programadores cometem frequentemente (DeMillo, 1978). Tais defeitos são propositalmente inseridos no software original por meio de uma mudança sintática simples, feitas para criar um conjunto de programas defeituosos. Cada um desses programas – chamados de *Mutantes* – contém uma mudança sintática distinta. Para avaliar a qualidade de um conjunto de dados de teste, esses mutantes são executados com os dados de teste, para avaliar se os defeitos semeados podem ser detectados. Análises atuais de trabalhos em Teste de Mutantes podem ser encontradas em (Jia e Harman, 2010), (Delamaro et al., 2007).

O processo de teste de um programa p usando a Análise de Mutantes é realizado do seguinte modo.

Um conjunto de *programas mutantes* p' é gerado por meio de pequenas mudanças sintáticas no programa original p . Após a geração dos programas mutantes, um conjunto de dados de teste T é executado no programa p e as saídas produzidas são checadas em relação ao esperado. Cada mutante p' é então executado com cada dado de teste t de T . Se o resultado da execução de p' é diferente do resultado da execução de p para algum dado de teste t , então o mutante p' é considerado *morto*, caso contrário, p' é considerado *vivo*.

O *escore de mutação* é uma medida de cobertura do critério análise de mutantes e denota o nível de adequação de um conjunto de dados de teste sob a perspectiva da Análise de Mutantes. Dado um programa P e um conjunto de dados de teste T , o escore de mutação $MS(P, T)$ é dado por:

$$MS(P, T) = \frac{DM(P, T)}{M(P) - EM(P)} \quad (2.1)$$

Sendo que $DM(P, T)$ é o conjunto de mutantes mortos pelo conjunto de dados de teste T ; $M(P)$ é o número total de mutantes gerados a partir do programa P ; e $EM(P)$ é o número de mutantes gerados que são equivalentes a P .

A teoria da análise de mutantes é baseada em duas hipóteses fundamentais: a hipótese do *Programador Competente* e a hipótese do *Efeito de Acoplamento*.

A hipótese do *programador competente*, como o nome sugere, estabelece que os programadores são competentes e, portanto, tendem a desenvolver programas que são próximos da versão correta. Deste modo, embora os programas possam conter defeitos, assume-se que esses defeitos são simples e podem ser corrigidos com pequenas alterações sintáticas. Com base nesta hipótese a análise de mutantes aplica apenas mudanças sintáticas simples, simulando defeitos inseridos por “programadores competentes”.

A hipótese do *efeito de acoplamento* estabelece que “dados de teste que distinguem todos os programas diferentes do programa correto apenas por erros simples são tão sensíveis, que eles também distinguem erros mais complexos”. Portanto, defeitos complexos estão acoplados a defeitos simples de uma maneira tal que dados de teste que detectam os defeitos simples irão detectar também um alto percentual de defeitos complexos.

No Apêndice A são detalhados o procedimento de teste na análise de mutantes, abordagens para a redução do custo de aplicação da técnica, níveis de teste e objetos de aplicação, além de descrever ferramentas de apoio, como a Proteum, utilizada na avaliação empírica descrita no Capítulo 5. São descritos os trabalhos: (Acree, 1980), (Howden, 1982), (Mathur, 1991), (DeMillo e Offutt, 1991), (DeMillo e Offutt, 1993), (Offutt et al., 1993),

(Delamaro e Maldonado, 1996), (Fabri et al., 1999 a), (Fabri et al., 1999 b), (Maldonado et al., 2000), (Barbosa et al, 2001), (Delamaro, Maldonado e Mathur, 2001) e (Jia e Harman, 2010).

2.7. Modelos de Detecção de Defeitos

A compreensão da técnica de teste propota nesta tese requer a consideração de mecanismos pelos quais os defeitos presentes no software acarretam falhas durante a execução. Esses mecanismos e, em especial as propriedades de *regiões de falha*, são descritos nesta seção.

Modelos foram desenvolvidos para caracterizar eventos e condições que ocorrem quando um software falha. Exemplos desses trabalhos incluem o modelo de fluxo de informação para detecção de defeitos (Ehrlich et al., 1991), (Thompson et al., 1993) e o modelo dinâmico de falhas de Voas (1992).

Durante a execução de um software que possui defeitos, é possível que todos os dados de teste sejam executados sem que o defeito seja revelado. Isto pode ocorrer se nenhum dado de teste exercitar o defeito; ou pode ocorrer se o defeito for exercitado, mas a falha do software não ocorrer. Este último caso, chamado de *Correção Coincidente* (Thompson et al., 1993) ou *Cancelamento* (Voas, 1992) é razoavelmente comum. Durante o teste, deseja-se exercitar os defeitos, evitar a Correção Coincidente, e fazer com que o software falhe. Para que a falha ocorra, é necessário que, ao se executar o software com um dado de teste, ocorram os eventos – segundo (Thompson et al., 1993):

1. O ponto no programa no qual existe um defeito deve ser executado;
2. Após a execução deste ponto, deve ser criado um estado intermediário incorreto, isto é, um comando ou expressão deve originar, ao ser executado, um resultado incorreto para alguma variável do programa;
3. Por meio de um fluxo de transferência de informação, o estado incorreto deve ser propagado e afetar alguma variável saída do programa.

Esses eventos são descritos com algumas variações em diversos trabalhos. Exemplos:

- Voas (1992) usa os termos: Execução, Infecção e Propagação para se referir aos eventos 1 a 3 (nesta ordem) em sua técnica dinâmica de estimar características de testabilidade de programas – o modelo PIE.
- No critério de teste Análise de Mutantes para se matar um mutante é necessário que, ao ser executado, ele produza uma saída diferente da produzida pelo programa original; portanto os eventos 1 a 3 devem ocorrer para que um programa mutante seja morto (DeMillo et al., 1978).
- Na abordagem Constraint Based Testing de geração de dados de teste para exercitar mutantes (DeMillo e Offutt, 1991), a *Condição de Necessidade* refere-se à execução do comando mutante e a criação do estado de execução errôneo (eventos 1 e 2); a *Condição de Suficiência* refere-se à produção de uma saída incorreta, requerida para se matar o mutante, e corresponde ao evento 3.

Do ponto de vista do domínio de entrada do programa, os dados de teste que satisfazem as condições 1, 2 e 3 acima constituem o Domínio de Falhas. Vários trabalhos abordam este conceito.

2.7.1. Domínios de Falha e Regiões de Falha

A caracterização de *Domínios de Falhas* e *Regiões de Falhas* é importante quando se trata de técnicas de teste que visam, em última análise, encontrar estas regiões.

Weyuker e Jeng definem pontos causadores de falha f que, em relação ao número de pontos d do domínio D , resultam em uma taxa de falhas do software ($\Theta = f/d$) que quantifica o quão provável é a falha do software ao ser executado (Weyuker e Jeng, 1991) – este trabalho é descrito detalhadamente no Apêndice C.

Frankl et al (1997) comparam técnicas de teste considerando a confiabilidade do software obtida após a eliminação de falhas encontradas no teste. Neste trabalho analítico o

conceito de defeito é crucial, no entanto, como o tratamento formal do conceito de defeito não existe segundo os autores, a definição do efeito dos defeitos (as falhas) é considerada ². A noção de Região de Falha é introduzida como “o conjunto de pontos de falha que é eliminado por uma correção do programa”.

Ammann e Knight (1988) na abordagem de tolerância a defeitos baseada em diversidade de dados (veja Apêndice C) definem um Região de Falha como “um domínio de falhas com a sua geometria”, sendo que domínio de falhas é o conjunto de pontos de entrada que causam a falha do programa.

Offutt e Hayes (1996) definem o Tamanho Semântico de um defeito como sendo o tamanho relativo do subdomínio de D para o qual a saída produzida é incorreta, em contraposição, o Tamanho Sintático de um defeito é o número de comandos que precisam ser alterados para que o programa fique correto. Implicações do conceito de tamanhos semântico e sintático são analisadas em relação às técnicas sementeira de defeitos e análise de mutantes e também em relação à testabilidade. O conceito de região de falha presente em Offutt e Hayes (1996) é próximo ao de Frankl et al (1997): “a região de falha para um defeito é a porção do domínio de entrada que faz com que o defeito resulte em uma falha”.

Um trabalho importante é o “Continuidade em Sistemas de Software” (Hamlet, 2002). Hamlet afirma que sistemas físicos como, por exemplo, alavancas, apresentam uma continuidade em seu comportamento. É possível testar a alavanca para alguns valores de força e ter confiança de que, se a alavanca não quebrou para estes valores ela também não quebrará com valores intermediários não testados.

No teste de sistemas mecânicos, usando a lógica acima, é requerido que o comportamento da função não varie de forma acentuada e satisfaça a *Condição de Lipschitz*. Esta condição garante que, ao se tomar pontos próximos a uma distância menor que b , o valor da função em análise não variará mais do que um limite B , ou seja, “a função

² Segundo (Frankl et al., 1997) um defeito é definido como “a parte do programa fonte que causa uma falha. Se um programa falha ele é alterado (ou corrigido) para que funcione corretamente. Esta alteração acaba caracterizando o defeito como, por exemplo, “operador relacional incorreto”. No entanto, na maioria das vezes, há várias diferentes maneiras de se corrigir um mesmo defeito (exemplo: defeitos por omissão de requisitos). Deste modo, a formalização deste conceito é complexa

não pode mudar de valor arbitrariamente em alguma vizinhança”. O teste de um sistema que satisfaça a *Condição de Lipschits* (representado por uma função Z) pode ser conduzido usando pontos $\{x_1, x_2, \dots, x_k\}$. Se estes pontos forem próximos o suficiente uns dos outros e os valores produzidos $Z(x_i)$ forem os esperados (corretos), garante-se que para os pontos intermediários não testados os valores $Z(x_i)$ não variarão mais do que B . Se uma variação de $Z(x_i)$ menor que B está dentro da “margem de segurança” para valores esperados, então garante-se que os resultados também serão corretos para esses pontos.

Em sistemas de software – ao contrário de alguns sistemas mecânicos – em geral não é possível garantir um comportamento contínuo e suave. Os comandos condicionais são a fonte da descontinuidade. Quando existem dois blocos de comandos (nós) situados após o *then* e após o *else* em um comando *if* ocorre o particionamento do domínio em dois subdomínios, cada um computando uma função diferente das variáveis usadas no bloco. Portanto, para dois pontos próximos, cada um pertencente a um subdomínio, os valores computados podem ser distantes um do outro.

Nota-se que as definições do conceito de *região de falha* apresentadas pelos vários autores são, de certo modo, mutualmente complementares e não se contradizem. Nesta tese a definição de Ammann e Knight (1988) é adotada por se enquadrar melhor na maioria dos contextos em que este conceito é referenciado.

2.7.2. Continuidade de Regiões de Falha

Uma continuidade em especial pode ser observada em software, a continuidade de regiões de falha. “Se o software falha para uma entrada então ele também deve falhar na vizinhança desta entrada. Analogamente, se o software funciona corretamente em um ponto, ele também deve funcionar corretamente na vizinhança deste ponto” (Hamlet, 2002). Se esta característica de continuidade se evidencia na maioria dos casos, este fato pode influenciar a eficácia das técnicas de teste em selecionar pontos pertencentes às regiões de falha. De todo modo, permanece o problema fundamental do teste de encontrar tais regiões de falha.

Trabalhos têm sido realizados para compreender a natureza das regiões de falha e avaliar a propriedade de continuidade dessas regiões. Ammann e Knight (1988) observam que as regiões de falha analisadas são localmente contínuas e variam de tamanho. Conforme descrito no Apêndice C na abordagem Diversidade de Dados uma re-expressão dos dados de entrada é realizada visando transformar uma entrada que provoca falha (dentro da região de falha) em uma outra para qual o software funcione corretamente (fora desta região). O objetivo é tolerar defeitos de software.

Bishop (1993) estuda características de falhas associadas a defeitos identificados em software de tempo real. Ao se avaliar experimentalmente a probabilidade de falhas em um software de controle de reator nuclear, identificou-se que, ao contrário do que sugere a teoria de confiabilidade tradicional, o software não falhava com uma probabilidade fixa por unidade de tempo. Para algumas distribuições de entrada a probabilidade média de falha por execução ($\bar{\lambda}$), a probabilidade de falha após n execuções sem falhas ($\lambda_s(n)$), e a probabilidade de falha após n execuções com falhas ($\lambda_f(n)$) apresentavam valores diferentes, indicando taxas de falhas não constantes. Esta variação de probabilidade de falha não ocorria para valores de entrada selecionados aleatoriamente, mas ocorria quando foram utilizadas entradas que faziam um “caminho aleatório”, no qual em cada execução os parâmetros de entrada eram alterados por um pequeno percentual. Este caminho aleatório simula uma situação típica nesse tipo de sistema, em que os valores de entrada não mudam muito de uma execução para a próxima.

A análise sobre este comportamento (taxas de falhas não constantes para o uso com o “caminho aleatório” no domínio de entrada) levou a evidências de que os defeitos formam “manchas” (*Blob Defects*) no domínio de entrada do programa, ou seja, os pontos defeituosos individuais encontram-se próximos uns dos outros (Bishop, 1993). Evidências e justificativas para os chamados “defeitos mancha” são fornecidas por análises empíricas e teóricas.

Evidências empíricas foram obtidas ao se “plotar” as regiões de falha, em projeções de duas dimensões do domínio de entrada, ao redor de pontos de falha detectados. Todos os 30 defeitos analisados ocupam regiões contínuas do domínio de entrada, fornecendo uma

“forte evidência experimental da existência de defeitos mancha”. Fatores que afetam essas regiões de falhas foram identificados:

- A dependência de variáveis de entrada: o defeito pode depender de apenas uma ou duas variáveis de entrada; os valores das demais variáveis não contribuem para a falha. Isto faz com que ao se fazer a projeção bidimensional das regiões de falha, mesmo defeitos com taxas de falhas baixas ocupem uma fração significativa do domínio de entrada em algumas dimensões. Isto é ilustrado na Figura 9 – original de Bishop (1993).
- A dependência de variáveis de entrada combinadas: alguns defeitos dependem de variáveis de entrada combinadas de acordo com alguma equação, nestes casos as regiões tendem a tomar formas cujos contornos são definidos pelas equações, muitas vezes oblíquos em relação aos eixos das variáveis. A Figura 10 – original de Bishop (1993) – mostra projeções bidimensionais de um defeito deste tipo.

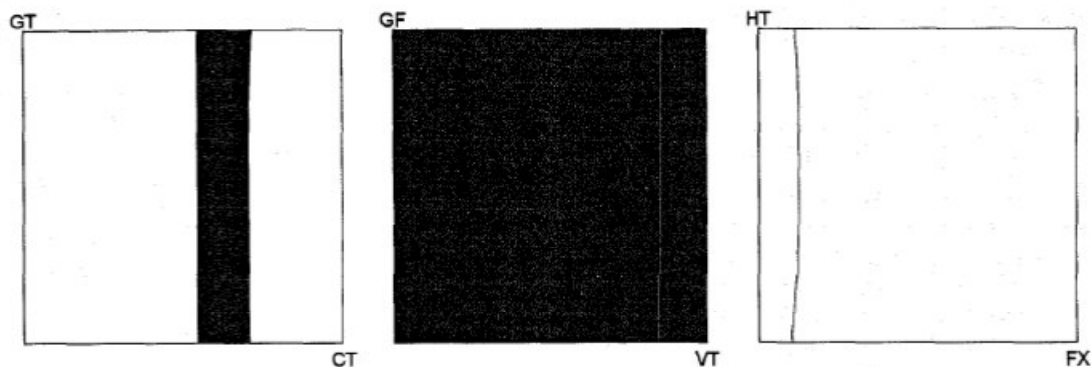


Figura 9. Projeções bidimensionais de uma região de falha dependente de um conjunto de variáveis de entrada

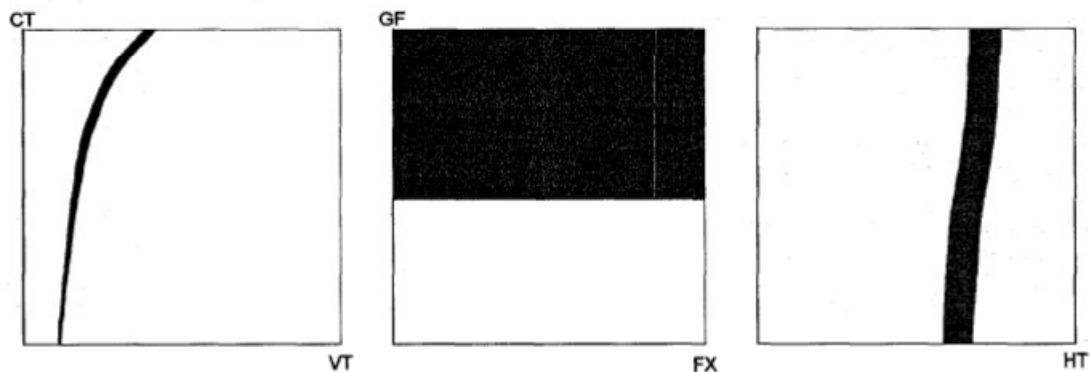


Figura 10. Projeções bidimensionais de uma região de falha dependente da relação entre variáveis de entrada

Evidências teóricas para os “defeitos mancha” vêm do fato de que qualquer computação em um programa pode ser tratada como uma composição de um conjunto de funções. Considerando que as funções computadas a partir da função defeituosa sejam funções suaves e contínuas, as regiões de falhas observadas também tendem a ser contínuas. Isto é, no caso de funções de mapeamento tipo um para um os estados errôneos são submetidos às funções computadas e tendem a se propagar ao longo da execução. No caso de programas que condensam a informação (funções de mapeamento muitos para um) um mascaramento do erro pode ocorrer, retirando partes da região original de erro.

As regiões de falha no domínio de entrada do programa podem ser definidas por meio de uma “projeção para trás” dos estados errôneos internos. À medida que a complexidade de programas aumenta as funções podem resultar em contornos muito distorcidos e dificultar qualquer previsão sobre as formas das regiões de falha no domínio de entrada. Além disto, há a possibilidade de que uma região interna errônea seja mapeada no domínio de entrada em diferentes regiões, criando múltiplos “defeitos mancha” ao invés de apenas um.

Dunham e Finelli (1990, 1991) realizaram estudos em um laboratório da NASA visando caracterizar aspectos fundamentais do processo de falhas em um software crítico de controle de voo. Dados sobre falhas do software foram gerados por meio de experimentos controlados que consideraram defeitos reais previamente identificados, e foram analisados segundo aspectos como: taxas de falhas; interação entre defeitos e padrões de regiões de

falha. Para avaliar a natureza das regiões do domínio de entrada associadas a defeitos foi realizado o seguinte procedimento. Para cada conjunto de valores de entrada em que o defeito resultou em falha do software, esses valores foram alterados ligeiramente e executados. O objetivo foi investigar o domínio de entrada adjacente às entradas causadoras de falha. Foi observado que na maioria dos casos um conjunto contínuo de pontos do domínio de entrada tende a provocar que o mesmo defeito produza falhas. Essas regiões são chamadas de “Cristais de Erros” (Error Crystals). A Figura 11 – original de Dunham e Finelli (1990) – mostra umas dessas regiões.

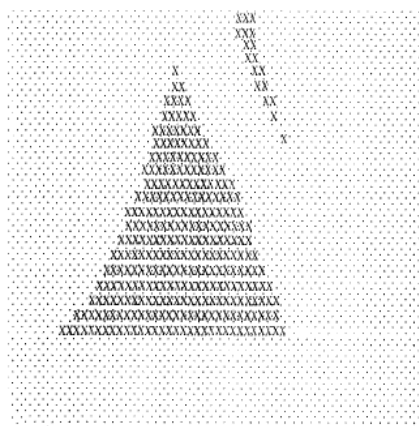


Figura 11. “Cristal de Erro” identificado em software da NASA

Schneckenburger e Mayer (2007) em um trabalho mais recente desenvolveram uma análise de quais aspectos de padrões de falhas são importantes em ser investigados e de como quantificar a forma e a localização de um padrão de falha. Três programas foram utilizados (*Bessj*, *Gammq*, *Exptint*) de Press et al (1992), todos os programas são numéricos e recebem parâmetros do tipo inteiro ou real. Para esses programas foram gerados programas mutantes (DeMillo et al., 1978) a serem utilizados na análise. Foi aplicada uma técnica de “análise de imagens” que marca pontos de falha com cor preta e pontos de funcionamento correto com cor branca. Foram geradas também imagens em tons cinza superpondo as diversas regiões de falha de cada programa (associadas a cada um dos mutantes). As imagens geradas foram analisadas em relação a características de

“clusterização” por meio de medidas como a compacidade (*compactness*)³ e foram comparadas a imagens geradas aleatoriamente. Foram identificados alguns padrões de alta compacidade (próximos de círculos e quadrados) e outros com menor compacidade como faixas. A conclusão geral é que a maioria dos padrões de falha são altamente compactos.

Chen et al (2005) no contexto da técnica Teste Aleatório Adaptativo propõe três padrões de região de falha: tipo Ponto, tipo Faixa e tipo Bloco. No padrão tipo Ponto as entradas causadoras de falha são pontos únicos, ou formam regiões de tamanho muito pequeno espalhadas no domínio de entrada. No padrão tipo Faixa a região de falha tem a forma de uma faixa fina. O padrão do tipo Bloco concentra entradas causadoras de falhas na forma de blocos. A Figura 12 mostram um domínio de entrada contendo falhas dos tipos Ponto, Faixa e Bloco.

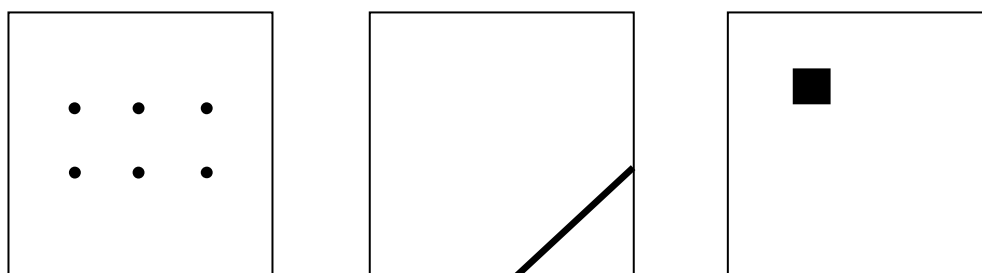


Figura 12. Padrões de Regiões de falha dos tipos: ponto, faixa e bloco.

As Figuras 13 (a), (b) e (c) mostram trechos de código contendo defeitos que originam falhas dos tipos Ponto, Faixa e Bloco, respectivamente (Chen et al., 2005). No trecho associado ao padrão tipo Ponto, o comando defeituoso é executado apenas quando X é múltiplo de 4 e Y é múltiplo de 6. Este tipo de defeito, ativado em situações muito específicas, é visto como um tipo menos comum. O trecho associado ao padrão tipo Faixa contém um defeito no comando condicional que faz com que a avaliação do predicado seja incorreta para valores de (X,Y) que fazem a expressão $(2*X-Y)$ assumir valores entre 10 e 18, originando assim uma faixa de valores que provocam a execução do comando defeituoso. Este tipo de defeito, associado a equações, inequações, ou operadores incorretos em predicados, é conhecido como “Defeito de Domínio” e motivou a definição da técnica

³ *Compactness* é definida como a razão entre a medida das bordas do padrão elevada ao quadrado e a área do padrão multiplicada por 4π . Outras medidas como o Número Euler também são utilizadas.

Teste de Domínios – *Domain Testing* (White e Cohen, 1980), (Clarke et al., 1982), (Zeil et al., 1992). No trecho associado ao padrão tipo Bloco o comando *if* estabelece faixas de valores para as variáveis (X,Y) que provocam a execução do comando defeituoso.

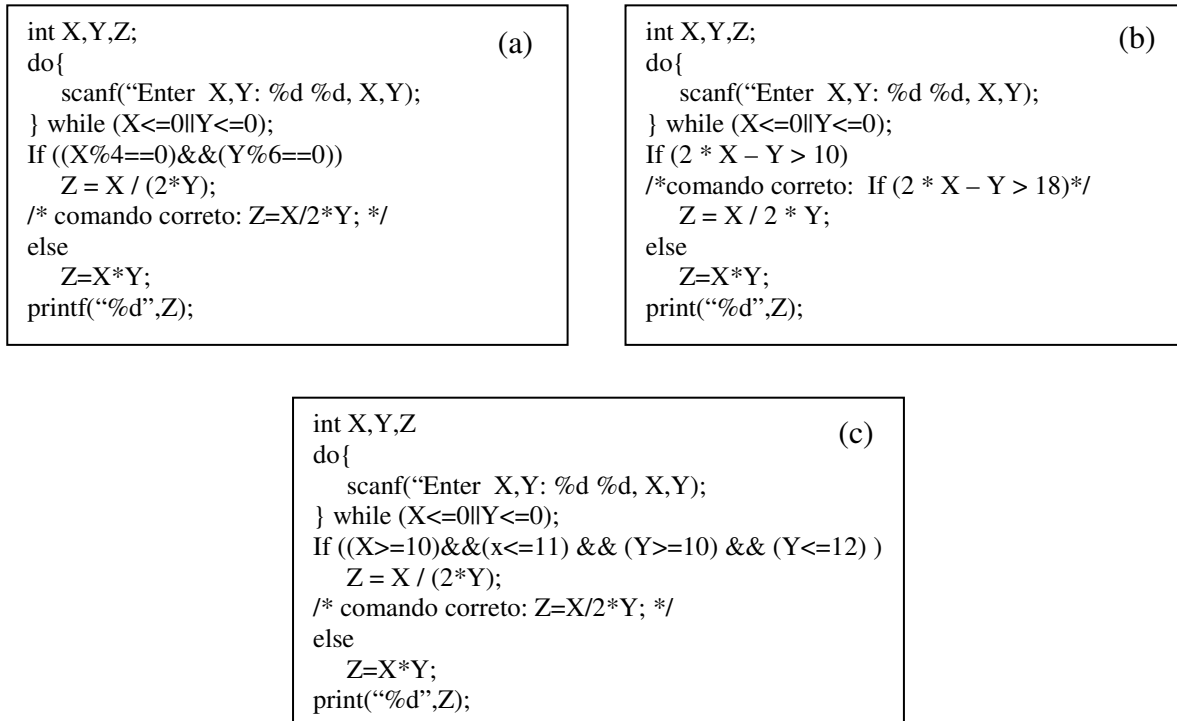


Figura 13. Trechos de programas que geram falhas dos tipos: ponto (a), faixa (b) e bloco(c)

Os Padrões de Região de Falha foram propostos como arquetípicos e podem ser utilizados para avaliar características de técnicas de teste. Por exemplo, esses padrões podem ser utilizados em simulações para avaliar a habilidade de revelar defeitos associados a cada padrão de região de falha. As regiões de falha associadas a defeitos “reais”, no entanto, podem combinar diferentes padrões em uma mesma região de falha, ou apresentar diferentes formas – conforme já descrito em (Dunham e Finelli, 1990), (Bishop, 1993).

2.8. Avaliação Empírica em Engenharia de Software

Esta seção apresenta brevemente conceitos associados à avaliação empírica em engenharia de Software. Este conteúdo estabelece a base conceitual da avaliação empírica descrita no Capítulo 5.

Basili (1996) identificou a necessidade de desenvolver uma abordagem experimental para a Engenharia de Software. Baseado na análise da natureza das avaliações empíricas em diversas áreas (ex: medicina, física, produção, etc.) o autor identifica que “a Engenharia de Software é uma ciência de laboratório. Ela possui um componente experimental para testar e refutar teorias e para explorar novos domínios”. O objetivo seria de experimentar técnicas, para ver como e quando elas funcionam, para compreender os seus limites e para aprimorá-las.

Kitchenham et al (2004) e Dyba et al (2005) destacam a importância de práticas baseadas em evidência na medicina e propõem a Engenharia de Software Baseada em Evidências – EBSE, sigla em Inglês. O objetivo da EBSE é “prover meios pelos quais as melhores evidências da pesquisa possam ser integradas à experiência prática e a valores humanos no processo de tomada de decisão relacionado ao desenvolvimento e manutenção de software”.

Barr et al. (1995) abordam especificamente experimentos computacionais para avaliar métodos heurísticos, o que inclui: definir objetivos do experimento, escolher medidas de desempenho, medidas de qualidade das soluções, avaliação de robustez, fatores que afetam desempenho, projeto, execução e análise de experimentos, dentre outros pontos. Os autores definem experimento como “um conjunto de testes, executados sob condições controladas, com um propósito específico: demonstrar uma verdade conhecida, checar a validade de uma hipótese, ou examinar o desempenho de alguma coisa nova”.

Em Shull et al (2008) vários colaboradores abordam temas avançados importantes em Engenharia de Software Empírica. Dentre os diversos temas, os mais relevantes no contexto desta tese são: Métodos de Simulação (Müller e Pfahl, 2008), Métodos Estatísticos e

Medição (Rosenberg, 2008), e Relatando Experimentos em Engenharia de Software (Jedlitschka et al., 2008).

Ali et al. (2010) apresentam uma revisão sistemática sobre a investigação empírica em trabalhos que abordam a geração de dados de teste usando técnicas de busca. Este trabalho provê direções sobre pontos importantes para a avaliação empírica em SBTS, dentre outros: a definição de objetivos e hipóteses, a seleção de programas, a seleção de medidas de custo e eficácia, avaliações de escalabilidade, definição de bases de comparação, e métodos de análise de dados.

2.9. Considerações Finais

Este capítulo abordou os conceitos essenciais em teste de software, com ênfase em técnicas e critérios de teste.

Conforme sugerem trabalhos empíricos, os diferentes critérios e técnicas tendem a revelar defeitos de natureza distinta no software. Desta forma, sugere-se que estratégias de teste estabeleçam o uso de várias técnicas. Neste caso, a diversidade das técnicas selecionadas atua a favor da eficácia do teste, visto que esta diversidade implica em uma natureza complementar em relação aos defeitos revelados. Esta conclusão é apresentada, por exemplo, em Littlewood et al (2000).

A seleção de critérios e técnicas de teste deve considerar o nível do teste em questão – unidade, integração, sistema, ou aceitação – visto que normalmente eles são mais adequados a níveis específicos. Outro ponto importante é que cada critério demanda um esforço diferente para ser aplicado, e exercita o software também com nível de rigor distinto. A seleção de critérios e técnicas deve se feita ponderando-se o que é um “teste suficiente”, dada a natureza do software em questão e os recursos disponíveis para o teste.

Um tema associado e complementar ao tema critérios e técnicas de teste é a geração de dados de teste, abordada no próximo capítulo.

Capítulo 3 – Geração de Dados de Teste

“Good tests kill flawed theories; we remain alive to guess again.”

Karl Popper

Este capítulo descreve inicialmente as principais abordagens para a geração de dados de teste, a execução simbólica e a técnica dinâmica. É apresentada a Engenharia de Software Baseada em Busca – SBSE, área em que meta-heurísticas (algoritmos de busca) são aplicadas para a solução de problemas de engenharia de software. O conceito de meta-heurística é apresentado e o mecanismo de funcionamento de algumas delas é descrito. Aborda-se então o Teste de Software Baseado em Busca – SBST, área em que meta-heurísticas são utilizadas para gerar dados de teste. Em seguida são descritos exemplos de trabalhos em SBST com ênfase no teste baseado em estrutura e no teste baseado em modelos.

Aborda-se então uma contribuição secundária desta tese. É fornecida uma visão geral de como um engenheiro de software pode definir uma abordagem SBST e são fornecidos alguns pontos chave a serem considerados nesta definição.

O Apêndice B descreve de forma mais extensa trabalhos que utilizam a execução simbólica e a técnica dinâmica. São oferecidos também alguns detalhes das soluções desenvolvidas em SBST.

3.1. Geração de Dados de Teste: Fundamentos e Trabalhos Iniciais

Conforme descrito no Capítulo 2, satisfazer um critério de teste significa exercitar todos os componentes requeridos pela aplicação do critério ao código ou à especificação em teste. Para que isto ocorra é necessário que, para cada elemento requerido, algum dado do conjunto de dados de teste o exercite.

A geração de dados de teste visa a selecionar pontos do domínio de entrada do programa para satisfazer um dado critério (cada ponto é um dado de teste). A tarefa de selecionar esses dados requer do testador conhecimentos específicos do critério utilizado além de uma cuidadosa análise da especificação do software e/ou do código do programa em teste. Isto faz com que a geração de dados de teste seja uma atividade de difícil realização e tenda a ter um alto impacto no custo final do teste.

A geração de dados de teste é dependente do critério de teste utilizado. Para cada critério é necessário identificar as características dos elementos requeridos pelo critério e as condições necessárias para que estes elementos sejam exercitados.

Exemplificando:

- No critério baseado na estrutura Todos os Nós os dados de teste devem ser selecionados de modo tal que o conjunto de caminhos executados exercite cada nó pelo menos uma vez;
- Os critérios baseados em análise de fluxo de dados definidos em (Rapps e Weyuker, 1985) requerem basicamente que dados de teste executem sub-caminhos livres de definição de cada nó contendo uma definição de uma variável, para nós contendo usos computacionais, e arcos contendo usos predicativos desta variável.

- Para que mutantes sejam mortos no critério Análise de Mutantes é necessário que sejam satisfeitas as condições de alcançabilidade, necessidade e suficiência para cada mutante gerado (DeMillo e Offutt, 1993).
- Critérios baseados em modelos de estados (ex: Máquinas de Estados Finitos) normalmente requerem que eventos de entrada exercitem as transições, ou sequências de transições, nos modelos.

Em uma perspectiva mais ampla, pode-se dizer que critérios e técnicas definem as características ou propriedades desejáveis de um conjunto de teste. Esta perspectiva, coerente com os critérios e técnicas exemplificados acima, explicita que a geração de dados pode focar em propriedades desejáveis de conjuntos de teste, mesmo que não diretamente relacionadas à cobertura de requisitos de teste. Por exemplo, a Geração Aleatória de dados baseada em um perfil de uso do software direciona a geração de dados com uma propriedade desejada (a aleatoriedade dos dados, ponderada pelo perfil de uso), sem estabelecer requisitos de teste a serem exercitados pelos dados de teste. A geração de dados de teste orientada à diversidade (próximo capítulo) é outro exemplo, busca-se neste caso gerar dados de teste com a propriedade de serem diversos.

Esforços têm sido conduzidos por diversos autores no sentido de automatizar a geração de dados de teste. A automação da atividade de geração de dados é em geral uma tarefa difícil e que envolve uma série de problemas complexos. Por exemplo: o problema de gerar dados para executar um determinado comando em um programa é equivalente ao problema da parada (“halting problem”) e, portanto, indecidível (Clarke, 1976), (Howden, 1975); a existência de caminhos não executáveis (caminhos para os quais não existe algum conjunto de valores de entrada do programa que os executem) pode fazer com que elementos requeridos pelos critérios sejam não exercitáveis. Para este problema (também indecidível) existirem soluções parciais, exemplos: (Frankl e Weyuker, 1988), (Malevris, 1990), (Hedley e Hennell, 1985), (Vergilio, 1992), (Bueno e Jino, 2000), (Forgács e Bertolino, 1997), (Bodík et al., 1997).

Duas abordagens se destacam na literatura para a automação da geração de dados: a execução simbólica (Clarke, 1976), (Howden, 1975); e a técnica dinâmica (Korel, 1990),

(Korel, 1996), (Ferguson e Korel, 1995), (Gallagher e Narasimhan, 1997). Estas abordagens podem ser utilizadas em diferentes contextos, tanto para a geração de dados para critérios baseados na estrutura quanto à geração de dados para exercitar modelos. Os primeiros trabalhos usando estas abordagens, porém, tratam especificamente da geração de dados para exercitar componentes estruturais, com ênfase a caminhos em programas. Exemplos dessas abordagens fornecidos à frente focam a geração de dados no teste estrutural.

Importante notar que a geração aleatória de dados pode também ser identificada como alternativa para a geração de dados para critérios estruturais e/ou baseados na especificação/modelos. Neste caso, o objetivo não é selecionar aleatoriamente dados de teste, executar o software e avaliar os resultados produzidos (realizar um Teste Aleatório); mas sim gerar aleatoriamente os dados com o objetivo de exercitar os elementos requeridos no teste. Isto é, nesta abordagem dados de teste são gerados aleatoriamente e executados no software, mas apenas aqueles que exercitam algum elemento requerido ainda não exercitado, são avaliados em relação aos resultados produzidos. Dados de teste que não exercitam algum elemento adicional são descartados. Uma limitação desta alternativa é que certos elementos requeridos são exercitados somente com dados de teste muito particulares, dificilmente gerados de forma aleatória. Exemplo: no teste estrutural, exercitar o valor verdadeiro para a condição `if (x==100)`, sendo `x` um inteiro.

A abordagem Simbólica e a abordagem Dinâmica buscam encontrar algum ponto do domínio de entrada que satisfaça a expressão simbólica associada ao elemento requerido que se pretende executar. No teste baseado em estrutura esta expressão, referida como expressão de caminho (Offutt, 1996), ou condição de caminho, (Gallagher e Narasimhan, 1997) reflete as condições necessárias para que um conjunto de dados de entrada provoque a execução do caminho. A expressão de caminho consiste na conjunção dos predicados encontrados ao longo do caminho e envolve variáveis do programa, constantes, operadores relacionais e lógicos. As variáveis de entrada do programa influenciam os predicados direta ou indiretamente através de “cadeias de interações de fluxo de dados” (Ntafos, 1984). Deste

modo, as expressões de caminho podem ser vistas como expressões simbólicas envolvendo as variáveis de entrada do programa.

Os trabalhos que utilizam execução simbólica buscam representar simbolicamente as condições para a execução de um dado caminho em função das variáveis de entrada do programa. Esta representação simbólica é utilizada por algoritmos que buscam dados de entrada que, ao satisfazer as condições, provoquem a execução do caminho pretendido.

A abordagem Dinâmica é baseada na execução real do programa, em métodos de minimização de funções e análise dinâmica de fluxo de dados. A partir de uma execução inicial, com dados de entrada selecionados aleatoriamente, procura-se aproximar iterativamente a execução real da desejada. Nas iterações os fluxos de execução e os valores de variáveis são monitorados, e técnicas de busca são utilizadas.

Trabalhos seguindo estas duas linhas são descritos a seguir. Edvardsson (1999) apresenta um *survey* sobre as abordagens simbólica e dinâmica.

3.2. Geração de Dados de Teste por Meio de Execução Simbólica

Execução simbólica e sua aplicação ao teste de programas não são ideias novas. Durante as duas últimas décadas, vários pesquisadores têm se dedicado a esse tema (Clarke, 1976), (Howden, 1975), (Boyer et al., 1975), (Ramamoorthy et al., 1976). Geralmente, ela é utilizada conjuntamente com a técnica baseada em caminhos e tem o objetivo de auxiliar a geração automática de dados de teste para um dado caminho ou conjunto de caminhos. Pode, em alguns casos, detectar caminhos não executáveis e, além disso, criar representações simbólicas para as variáveis de saída na execução de um caminho. Estas representações podem ser comparadas com representações simbólicas das saídas esperadas, facilitando a detecção de defeitos. A execução simbólica é utilizada também em outras áreas da computação, tais como a depuração de programas (Clarke e Richardson, 1983).

Técnicas de execução simbólica derivam expressões algébricas que representam a execução de um dado caminho (ou conjunto de caminhos). O resultado é uma expressão

simbólica chamada computação do caminho que representa as variáveis de saída em termos das variáveis de entrada, e uma expressão, chamada condição do caminho, que representa condições para que o caminho seja executado (Clarke, 1976).

A condição do caminho é dada pelo conjunto de restrições associadas a todos os predicados encontrados ao longo do mesmo. 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.

A Figura 14 ilustra a ideia geral desta técnica. Nesta figura os valores simbólicos a , b e c são atribuídos às variáveis x , y e z , respectivamente. Estes valores são processados simbolicamente de modo que as variáveis internas e os predicados do programa sejam representados em função das variáveis de entrada (no exemplo a variável t é equivalente a $a + b$ e o primeiro predicado equivalente a $c > b$). A conjunção dos predicados do caminho que se deseja executar representa a condição que precisa ser satisfeita para que o caminho seja executado. No exemplo $(c > b)$ e $(b > a)$ e $((a + b) < c)$ determina a condição suficiente para que os três predicados existentes sejam satisfeitos.

main()	
scanf("%d%d%d",&x,&y,&z);	$\begin{matrix} a & b & c \\ \downarrow & \downarrow & \downarrow \\ x & y & z \end{matrix}$
if (z > y)	[c > b]
{	
if (y > x)	[b > a]
{	
t = x + y;	[t = a + b]
if (t < z)	[(a + b) < c]
{	
...	

Figura 14. Geração de dados de teste por meio de execução simbólica

Algumas limitações da execução simbólica são descritas a seguir. Nem sempre é possível determinar se existe uma solução que satisfaça a condição do caminho, ou seja, determinar se o caminho é ou não executável. Também não é garantido que para todos os casos nos quais existe alguma solução ela será descoberta. O tratamento de laços tende a ser

complexo; quando o número de iterações no laço não é determinado é necessário deduzir os valores das variáveis no laço, o que pode gerar relações de recorrência de difícil solução (Offutt, 1996). Existem problemas quando há referências a variáveis compostas (vetores e matrizes) e a apontadores; o problema é identificar a qual variável se faz referência. O tratamento de chamadas de outros módulos por processamento simbólico tende a gerar equações muito extensas e complexas (Howden, 1977).

O Apêndice B apresenta outros trabalhos que utilizam a execução simbólica.

Buscando um melhor tratamento desses aspectos foi proposta a técnica dinâmica de geração de dados de teste, abordada a seguir.

3.3. Abordagem Dinâmica Para a Geração de Dados de Teste

Conforme descrito anteriormente, a abordagem dinâmica é baseada na execução real do programa, em métodos de minimização de funções e análise dinâmica de fluxo de dados. Embora proposta inicialmente para a geração de dados no teste estrutural (baseado no código fonte), esta abordagem pode ser aplicada em outros problemas, como na geração de dados para exercitar modelos (por exemplo, para exercitar a estrutura de modelos).

A Figura 15 ilustra esta técnica. Valores gerados aleatoriamente são atribuídos às variáveis de entrada e o programa é executado. Caso o fluxo de execução desvie do pretendido em algum predicado, técnicas de otimização são aplicadas para modificar os valores de entrada visando provocar a execução do fluxo de controle desejado. Exemplificando: caso seja desejada a satisfação do predicado ($z > y$) e isto não ocorra na execução, é associada ao predicado uma função $E1 = y - z$. Esta função permite apurar o “erro” que fez com que o predicado não fosse avaliado como o desejado. O valor E1, obtido por meio do monitoramento da execução, pode ser utilizado para direcionar as mudanças nas variáveis de entrada visando à satisfação do predicado em questão; mudanças nessas variáveis que minimizam o valor de E1 estão na “direção correta”, isto é, podem levar à satisfação do predicado.

No programa da Figura 15 caso as variáveis de entrada tenham os valores $x = 2$, $y = 5$ e $z = 3$ o predicado $(z > y)$ não será satisfeito e será computado o valor $E1 = y - z = 2$. Para os valores $x = 2$, $y = 5$ e $z = 4$ o valor computado para $E1$ será menor ($E1 = y - z = 1$), o que significa que o predicado está mais “próximo de ser satisfeito”. Para os valores $x = 2$, $y = 5$ e $z = 6$ o predicado $(z > y)$ será satisfeito, assim como o predicado $(y > x)$; o predicado $(t < z)$, por outro lado, não será satisfeito. Será necessário, portanto realizar uma nova busca por valores para x , y e z que minimizem $E3$ a fim de satisfazer o predicado $(t < z)$. Deve-se notar que, na tentativa de satisfazer o terceiro predicado, existe a restrição de que os dois predicados anteriores devem continuar sendo satisfeitos; ou seja, as mudanças nas variáveis de entrada não devem mudar as avaliações dos predicados anteriores.

```
main()
scanf("%d%d%d",&x,&y,&z);
if ( z > y )          [ E1 = y - z ]
{
    if ( y > x )      [ E2 = x - y ]
    {
        t = x + y;
        if ( t < z )  [ E3 = t - z ]
        {
            ...
        }
    }
}
```

Figura 15. Geração de dados de teste usando a abordagem dinâmica

A seguir são descritos brevemente os trabalhos iniciais que utilizam a abordagem dinâmica.

A técnica dinâmica foi proposta inicialmente por Miller e Spooner (1976). Os autores propõem a solução do problema de geração de dados de teste como um problema de otimização numérica. Os comandos de decisão do programa são trocados por comandos chamados de restrições de caminho, que medem o quão próxima cada decisão está de ser satisfeita. O programa toma a forma de uma única linha de código com uma série de restrições cujos erros devem ser minimizados. Variáveis de entrada cujo tipo não seja real

devem ser atribuídas pelo testador. A busca por dados de entrada que executem o caminho, para as variáveis do tipo real, é realizada por algoritmos de busca direta.

Korel (1990) desenvolveu um protótipo denominado TESTGEN, compatível com um subconjunto do Pascal, que utiliza a técnica no intuito de encontrar valores para as variáveis de entrada do programa tais que um determinado caminho seja executado. Dados reais são atribuídos às variáveis de entrada e o fluxo de execução do programa é monitorado. Se um ramo incorreto foi tomado, métodos de minimização de funções são utilizados para determinar valores para as variáveis de entrada para os quais o ramo correto seria tomado. A abordagem considera predicados simples e lineares da forma $E_1 \text{ op } E_2$. Todo predicado pode ser transformado na forma $F \text{ rel } 0$, onde $\text{rel} \in \{<, \leq, =\}$. Esta abordagem permite o tratamento de estruturas do tipo vetor.

O problema inicial é determinar um conjunto x^0 de valores de entrada que provoquem a execução de um dado caminho $P = (n_1, \dots, n_i, \dots, n_m)$. Korel diz que esse problema pode ser reduzido a uma sequência de sub-objetivos, em que cada sub-objetivo será resolvido utilizando-se técnicas de minimização de funções. Se P é executado com o conjunto x^0 de valores de entrada, então x^0 é a solução para o problema de geração. Se não foi executado um outro caminho T e x^0 não é a solução. Neste caso $P_1 = (n_1^p, n_2^p, \dots, n_i^p)$ é o mais longo sub-caminho de T , tal que $(n_1^p = n_1, \dots, n_i^p = n_i)$, e o caminho correto não foi executado porque o predicado associado ao arco (n_i, n_{i+1}) não foi avaliado como se esperava. Então o objetivo é encontrar valores de entrada x^1 tais que $F_i(x) \text{ rel}_i 0$ seja satisfeita, provocando a execução do ramo (n_i, n_{i+1}) como pretendido. A função $F_i(x)$ pode ser minimizada até que ela se torne negativa (ou 0 dependendo de rel_i). Uma vez resolvido o primeiro sub-objetivo, outros deverão ser resolvidos até que a solução do objetivo principal seja encontrada, ou ainda, até que se decida que o sub-objetivo não poderá ser resolvido (nesse caso a pesquisa falha).

A busca dos valores é feita pelo Método da Variável Alternativa, que consiste em minimizar com respeito a somente uma variável de entrada por vez. Se, considerando uma variável x_i , a função $F_i(x)$ não se torna negativa, as demais variáveis de entrada são consideradas, uma por vez. Neste processo são feitos movimentos exploratórios a fim de

definir como alterar a variável considerada para que $F_i(x)$ diminua. Se esses movimentos indicam uma direção a ser tomada, uma mudança de maiores proporções (aumento ou decremento da variável x_i), é executada. A pesquisa falhará se todas as variáveis forem analisadas e a função $F_i(x)$ não puder ser decrementada. É mostrada também uma adaptação dessas ideias para programas que manipulam estruturas de dados dinâmicas.

Esta técnica foi estendida por Ferguson e Korel (1996) de forma a incorporar a seleção de caminhos no processo de geração de dados. São utilizadas informações sobre dependência de dados para determinar comandos do programa que afetam a execução de outros. Deste modo, são definidas “sequências de eventos” (sequências de nós) que devem ser executadas antes de se atingir um dado predicado, de forma que o algoritmo de busca tenha maior chance de satisfazê-lo.

Abordagem Dinâmica Para a Geração de Dados de Teste: Sumário

Desta forma, na abordagem dinâmica a interferência de chamadas de outros módulos e variáveis compostas nos predicados (aspectos críticos para o processamento simbólico) é apurada a partir de valores reais das variáveis na execução e não através de expressões simbólicas. Isto faz com que tais problemas sejam contornados, deixando a tarefa de busca dos dados para executar o caminho para o algoritmo de busca utilizado.

O Apêndice B apresenta outros trabalhos que utilizam a abordagem dinâmica.

3.4. Engenharia de Software Baseada em Busca

A aplicação de algoritmos de busca (referidos também como meta-heurísticas, ou como técnicas de otimização) para a solução de problemas de engenharia de software é referida como Engenharia de Software Baseada em Busca. (*Search Based Software Engineering* – SBSE). Embora a expressão *Search Based Software Engineering* tenha sido cunhada em 2001 (Harman e Jones, 2001), trabalhos anteriores a esta data já aplicavam

técnicas de busca em problema de engenharia de software, exemplos: (Miller e Spooner, 1976), (Korel, 1990).

Trabalhos nesta área de pesquisa reformulam problemas da engenharia de software como problemas de otimização baseada em busca. Soluções ótimas ou sub-ótimas para problemas são procuradas em um espaço de soluções candidatas, em geral tendo como guia uma função objetivo que distingue as soluções melhores das piores. De certo modo a área SBSE é parte de um campo mais abrangente: a aplicação de mecanismos de inteligência computacional para o tratamento de problemas em engenharia de software (Pedrycz e Peters, 1998), (Pedrycz, 2002).

Mantere e Alander (2005) apresentam uma revisão da “Engenharia de Software Evolutiva”. Harman et al. apresentam revisões completas sobre a área SBSE, classificando os trabalhos segundo o tipo de problema da engenharia de software tratado (Harman, 2007), (Harman et al., 2009). Vergilio et al (2011) apresentam uma revisão de trabalhos de autores brasileiros na área SBSE.

Vários problemas de engenharia de software são abordados com o uso de meta-heurísticas. Subáreas da engenharia de software em que meta-heurísticas são utilizadas incluem: análise de requisitos e especificação de software, projeto de software, verificação e checagem de modelos, manutenção, refatoração, e gerenciamento de projetos. Trabalhos relacionados à subárea teste de software serão detalhados nas próximas seções.

Conforme destacado nesta seção a aplicação de meta-heurísticas é uma característica fundamental de trabalhos da área SBSE. A seguir é feita uma descrição dessas técnicas.

3.5. Meta-heurísticas

Meta-heurísticas são métodos ou algoritmos aproximados comumente aplicados para tratar problemas complexos, como problemas de otimização combinatória. Nestes problemas, em geral busca-se encontrar boas soluções, sob algum aspecto, em um espaço (potencialmente grande) de possíveis soluções. Algoritmos exatos garantem a obtenção de

soluções ótimas, mas, para o caso de problemas NP-Completo, encontrar a solução ótima pode requerer uma computação cujo custo aumenta a uma taxa maior que polinomial, com o tamanho do problema, tornando esta opção inviável em termos práticos. Este fato torna relevante a utilização de métodos aproximados, que permitam obter soluções de boa qualidade (idealmente ótimas) com um custo computacional factível. Blum e Roli (2003) apresentam uma visão geral e análises sobre meta-heurísticas.

O termo meta-heurística é derivado da composição de duas palavras de origem grega. A palavra “Heurística” deriva de um verbo que significa “encontrar”. O prefixo “Meta” significa “além, em um nível superior”. Meta-heurística pode ser definida como um “processo iterativo que guia uma heurística subordinada por meio da combinação inteligente de diferentes conceitos para explorar o espaço de busca (...)”. Propriedades principais que caracterizam as meta-heurísticas são descritas por Blum e Roli: são estratégias que guiam o processo de busca; exploram eficientemente o espaço de busca para encontrar soluções próximas do ótimo; podem incorporar desde busca local simples, até processos de aprendizado complexos; são aproximadas e usualmente não determinísticas; incorporam mecanismos para evitar ficarem “presas” em áreas confinadas no espaço de busca; permitem um nível de descrição abstrato; não são específicas para determinados problemas; e podem fazer uso de conhecimento específico do domínio na forma de uma heurística, que é controlada por uma estratégia de nível superior (Blumm e Roli, 2003).

De modo geral, o processo de resolução de problemas por meio de meta-heurísticas (ou Métodos de Busca) corresponde à tomada de uma sequência de ações que levam a um desempenho desejado, ou melhore o desempenho de *soluções candidatas* (cada solução candidata é uma possível solução para o problema). Este processo de resolução de problemas é denominado uma *busca* (Von Zuben e De Castro, 2002).

Uma meta-heurística tem como entrada um problema e produz como saída uma solução. Durante uma busca, uma ou mais soluções candidatas são utilizadas e manipuladas. Todas as soluções candidatas são representadas (ou codificadas) de um modo definido e situam-se em um espaço de busca (ou espaço de soluções), cuja dimensão é definida pelo número de atributos (ou variáveis) representados nas soluções candidatas.

Soluções candidatas são avaliadas por uma função de avaliação que retorna a qualidade relativa da solução, considerando o objetivo da busca (tipicamente um objetivo é maximizar valores da função de avaliação, ou minimizar valores desta função). A função de avaliação estabelece uma superfície de resposta (também chamada superfície de *fitness*, ou superfície de adaptação), com picos, platôs e vales. A busca consiste, portanto em encontrar regiões de picos na superfície de adaptação, supondo a busca por valores máximos, ou regiões de vales, caso contrário (Von Zuben e De Castro, 2002).

A seguir, são descritas inicialmente as meta-heurísticas: Subida de Encosta, Recozimento Simulado e Algoritmo Genético. Estas três meta-heurísticas apresentam grande prevalência na área SBSE (Harman et al., 2009). São descritos também os algoritmos de Seleção Clonal e a Otimização Baseada em Enxame de Partículas. Outras abordagens são descritas de forma resumida em seção posterior.

3.5.1. Subida de Encosta

A meta-heurística Subida de Encosta (SE), ou Busca Local, focaliza a atenção em uma vizinhança específica do espaço de busca (Michalewicz e Fogel, 2000). O procedimento da busca com SE é formado pelos seguintes passos:

1. Selecione uma solução do espaço de busca e avalie o seu mérito. Defina esta solução como solução atual;
2. Aplique uma transformação na solução atual para gerar uma nova solução e avalie o seu mérito;
3. Se a nova solução é melhor do que a solução atual, torne esta solução a solução atual, caso contrário, descarte a nova solução;
4. Repita os passos 2 e 3 até que nenhuma transformação melhore a solução atual.

A SE pode variar em como uma nova solução é gerada a partir da solução anterior. Normalmente, uma transformação na solução atual cria uma nova solução na vizinhança daquela, no espaço de busca. Deste modo, a solução atual estabelece um ponto a partir do qual melhorias são buscadas. Movimentos são realizados apenas para soluções melhores do

que a solução atual, sendo que a exploração da vizinhança em cada passo pode ser feita investigando apenas uma opção, ou explorando a vizinhança e escolhendo a melhor opção, dentre as várias investigadas.

O desempenho da SE pode ser limitado, devido à tendência da busca ficar “estacionada” em pontos sub-ótimos do espaço de busca (mínimos locais ou máximos locais, dependendo do objetivo da otimização). O sucesso ou fracasso em encontrar soluções ótimas depende essencialmente do ponto inicial da busca; uma variação desta técnica consiste em realizar diversas buscas, com pontos iniciais diferentes, e selecionar a melhor solução obtida nessas diversas buscas (Von Zuben e De Castro, 2002).

3.5.2. Recozimento Simulado

A meta-heurística *Simulated Annealing* (Recozimento Simulado – RS) é semelhante à Subida de Encosta. No entanto, enquanto a SE realiza movimentos apenas para soluções da vizinhança que sejam melhores que a atual, o RS pode realizar movimentos, de forma controlada, também para soluções da vizinhança que não apresentem uma melhora da função objetivo, criando a possibilidade de escapar de pontos ótimos locais.

O Recozimento Simulado, apresentado por Kirkpatrick et al., é inspirado em uma analogia com termodinâmica, mais especificamente com a maneira com que os líquidos congelam e se cristalizam (Kirkpatrick, 1984), (Kirkpatrick et. al, 1983), (Eglese,1990). Este trabalho tem origem na estratégia para a determinação de estados de equilíbrio em uma coleção de átomos a uma dada temperatura (Metropolis et al., 1953) (um problema de termodinâmica estatística, ramo da física que busca apresentar previsões teóricas sobre o comportamento de sistemas baseados nas leis que governam seus átomos).

Em altas temperaturas, as moléculas de um líquido se movem quase livremente. Se o líquido for resfriado de forma gradual as moléculas perdem mobilidade e tomam a forma de um cristal completamente organizado – a energia mínima global do sistema. O processo de congelamento é guiado por um escalonamento de resfriamento que controla o decaimento da temperatura do sistema.

O algoritmo inicia gerando um ponto inicial para a busca e ajustando o parâmetro temperatura (T). A cada iteração, uma nova solução s' é gerada a partir do ponto anterior s . s' é aceita como nova solução corrente dependendo dos valores objetivo $f(s)$ e $f(s')$ e de T. Considerando o objetivo de maximizar f , s' substitui s se $f(s') > f(s)$. Caso contrário ($f(s') \leq f(s)$) s' é aceita como nova solução corrente com uma probabilidade que é função de T e de $f(s) - f(s')$. A probabilidade de aceitação p da solução é dada por:

$$p = \frac{1}{1 + e^{\frac{f(s) - f(s')}{T}}} \quad (3.1)$$

Quanto maior o valor $f(s) - f(s')$, menor a chance do movimento para s' . Isto é, quanto maior a piora nas avaliações, menor a chance do movimento para uma solução pior ser aceito. Quanto maior o valor do parâmetro temperatura (T), maior a chance de movimentos para soluções piores. Durante a busca, T é atualizado de acordo com um cronograma de resfriamento, que reduz progressivamente a temperatura. Nos estágios iniciais da busca, valores altos de T enfatizam a exploração do espaço de busca, tendendo a uma busca aleatória. Em estágios mais avançados da busca, valores mais baixos para T levam o comportamento do RS a tornar-se próximo do comportamento da Subida de Encosta, aceitando apenas movimentos para soluções melhores.

A Figura 16 descreve a estrutura do RS. O laço interno realiza um determinado número de iterações, definida pela condição de término. Em cada iteração deste laço, é feita a geração de um ponto s' na vizinhança do ponto corrente (solução corrente s) que é sujeito à aceitação como novo ponto corrente, conforme já descrito. No laço externo, é realizado o decréscimo de temperatura, definido pelo escalonamento de resfriamento. A busca se encerra quando satisfeito o critério de parada, por exemplo, ao se encontrar uma solução desejada, ou se um número máximo de iterações for atingido.

```

Procedimento Recozimento Simulado
Início
t←0
iniciar T,
selecionar aleatoriamente um ponto corrente s
avaliar s
repetir
  repetir
    selecionar novo ponto s' na vizinhança de s
    se f(s) < f(s') então s ← s'
    senão se random[0,1) < e $\frac{-(f(s)-f(s'))}{t}$ 
      então s ← s'
  até (cond_termino)
t←g(T,t)
t ← t+1
até (critério_parada)
fim

```

Figura 16. Recozimento Simulado (versão para maximização)

3.5.3. Algoritmo Genético

Algoritmos Genéticos (AGs) são algoritmos de busca inspirados nos mecanismos da evolução, seleção natural e da genética (Holland, 1992), (Goldberg, 1989), (Srinivas e Patnaik, 1994), (Michalewicz, 1996). Inicialmente propostos por John Holland, tais algoritmos trabalham com uma população de indivíduos que sofrem processos de seleção, na presença de operadores que induzem variações. Uma função de ajuste – *fitness* (ou função objetivo) é utilizada para avaliar os indivíduos e determinar a chance de sucesso de cada um na seleção.

Estes algoritmos refletem o fato de que, na natureza, a competição por recursos escassos (alimentação, espaço, parceiros para acasalamento) faz com que indivíduos mais bem adaptados dominem os mais fracos. Este fenômeno, denominado “sobrevivência do mais apto”, estabelece a seleção natural, elemento essencial na evolução das espécies. Neste mecanismo, as características que definem unicamente cada indivíduo determinam a sua capacidade de sobrevivência. Cada característica relaciona-se a um gene. Conjuntos de

genes controlando as diversas características formam os cromossomos. A seleção natural enfatiza a sobrevivência dos melhores indivíduos, mas também, implicitamente, enfatiza a sobrevivência dos melhores genes. Isto é, indivíduos mais bem adaptados têm mais chances de sobreviver e influenciar gerações posteriores, estendendo suas características genéticas a essas gerações.

A Figura 17 descreve o funcionamento básico de um Algoritmo Genético. Inicialmente, é gerada uma população de indivíduos – potenciais soluções para um problema (possivelmente de forma aleatória). Estas soluções são submetidas a um processo de avaliação, que utiliza uma função (função objetivo ou função de *fitness*) e associa a cada solução um valor que mede a sua qualidade (referido como valor objetivo, ou valor de *fitness*). É feita uma seleção de soluções para a próxima iteração, baseada no mérito de cada solução, e são aplicados operadores genéticos, que manipulam as soluções, transformando-as de acordo com um processo aleatório, mas, ao mesmo tempo, estruturado. Após selecionadas e possivelmente alteradas, as soluções candidatas – os indivíduos da população – constituem uma nova geração, entrada para uma nova iteração do AG.

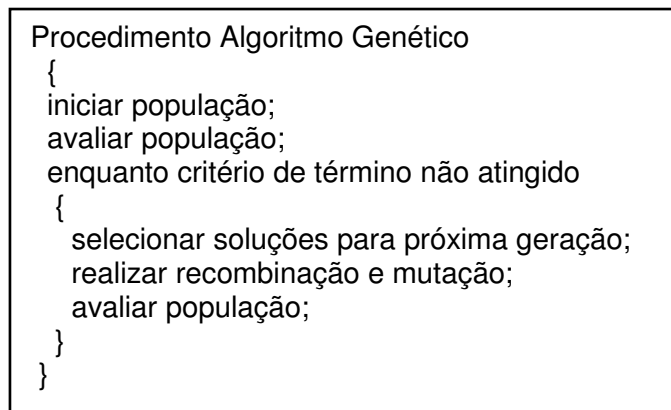


Figura 17. Algoritmo Genético

Em geral, o processo de seleção realizado em cada iteração associa a cada solução candidata uma chance de seleção proporcional à sua qualidade relativa. Ou seja, soluções com maiores valores para a função objetivo são privilegiadas (Srinivas e Patnaik, 1994). A evolução ocorre a partir da existência de uma diversidade de indivíduos (soluções

candidatas) competindo entre si e compartilhando material genético na geração de descendentes.

O funcionamento dos AGs está baseado na ideia de que, com o passar das gerações, as boas soluções tendem a compartilhar partes de seus cromossomos. Essas partes constituem os esquemas, que são padrões descrevendo subconjuntos de soluções similares. Esquemas que levam às soluções melhores são identificados progressivamente ao longo da busca e utilizados como “blocos de construção” das novas soluções. Deste modo, ocorre uma exploração de informação histórica para direcionar a busca para novos pontos no domínio, que apresentem expectativa de melhorias tendo em vista a solução do problema (Holland, 1992).

Alguns elementos importantes para a aplicação de AGs podem ser destacados:

Codificação: a representação em forma codificada das variáveis envolvidas no problema de otimização é importante e influencia o desempenho do AG. Em geral, busca-se mapear os parâmetros do problema em um único “string”, representando uma solução candidata. O Algoritmo Genético Simples trabalha com uma população de *strings* binários (Goldberg, 1989). Outras representações, como variáveis do tipo real ou do tipo inteiro concatenadas, também podem ser utilizadas, assim como listas de adjacências (Srinivas e Patnaik, 1994).

Função objetivo (fitness) e mecanismo de seleção: a função objetivo reflete o objetivo da busca, provendo um mecanismo para avaliar e qualificar cada indivíduo. Intuitivamente, esta função pode ser vista como uma medida de qualidade ou utilidade da solução codificada no indivíduo. Segundo Goldberg (1989), a função objetivo deve ser um valor não negativo que reflita o mérito da solução que se deseja maximizar. Caso esta restrição não seja atendida, é necessário definir mapeamentos que transformem a função objetivo da busca, tornando-a adequada. Pode-se também normalizar o valor da função objetivo, gerando valores entre 0 e 1.

Um mecanismo de seleção amplamente utilizado é a *seleção proporcional*. Soluções mais aptas têm uma maior probabilidade de contribuir com um ou mais descendentes na

próxima geração. Isto ocorre porque a alocação de descendentes é baseada na razão entre a aptidão do indivíduo e a média da aptidão da população, ou seja, a chance de o indivíduo ser selecionado é definida pela relação f_i/F , sendo f_i a aptidão do indivíduo e F a aptidão média da população.

Mecanismos de *escalonamento* ou a definição de *ranques* podem aperfeiçoar estágios iniciais da busca, quando super indivíduos podem dominar prematuramente a população, e em estágios finais, quando os melhores indivíduos destacam-se muito pouco em relação à média.

O *elitismo* visa preservar os melhores indivíduos da população. Na seleção proporcional, não é garantido que a melhor solução para o problema (associada ao melhor indivíduo) sobreviva para a próxima geração. A seleção elitista inclui obrigatoriamente o melhor indivíduo da população na geração subsequente. Isto evita que boas soluções sejam perdidas devido ao caráter aleatório da seleção proporcional (Goldberg, 1989).

Na *seleção por torneio* k indivíduos da população são escolhidos aleatoriamente na população corrente ($k \geq 2$). O melhor entre esses indivíduos (com maior valor de aptidão) é selecionado para compor a população da próxima geração. O torneio é repetido várias vezes para formar a população da próxima geração.

Operadores genéticos: além do operador de seleção, já descrito, operadores genéticos comuns são a recombinação e a mutação. A operação de recombinação (*crossover*) é feita após a seleção, e promove a troca de material genético entre dois indivíduos da população (resultante da reprodução sexuada). A ideia da recombinação é que características positivas de dois indivíduos podem ser combinadas, gerando uma descendência cujo nível de ajuste é potencialmente superior ao de ambos os pais. A probabilidade de que o operador seja aplicado é um parâmetro de controle do Algoritmo Genético (Michalewicz, 1996).

Vários operadores de recombinação foram desenvolvidos, dependendo do problema e da codificação específicos. Por exemplo, a *recombinação de ponto único* (também chamada simples) realiza a troca de valores codificados em dois *strings* soluções (indivíduos) de

forma que uma posição p ao longo do *string* é selecionada aleatoriamente. Novos *strings* são gerados pela troca de todos os valores (genes) encontrados a partir de p (inclusive) até o final do *string*. A *recombinação uniforme* realiza a troca de valores (genes) em dois indivíduos. Para cada gene do indivíduo pai 1 é decidido probabilisticamente se o gene fará parte do filho 1, ou do filho 2. A decisão contrária é tomada para o indivíduo pai 2. Portanto, enquanto a *recombinação de ponto único* realiza a troca de segmentos de genes para gerar os descendentes, a *recombinação uniforme* realiza a troca de genes (separadamente, não em segmentos) para gerá-los.

A operação de mutação é aplicada após a recombinação e consiste em uma alteração ocasional (com baixa probabilidade associada) e aleatória do valor de uma posição do *string* que representa uma solução candidata. Geralmente, a mutação de um gene não afeta a probabilidade de mutação de outros genes. Este operador tem a função de espalhar os indivíduos pelo espaço de busca, promovendo uma maior exploração. Por exemplo, caso em uma população o valor associado a um gene seja o mesmo para todos os indivíduos, a mutação (ao contrário da recombinação) pode gerar um valor diferente para este gene.

Os operadores genéticos não se restringem aos mencionados acima. Outros operadores consideram dominância, reordenação de código, inversão, duplicação entre cromossomos, segregação. Operadores orientados à população permitem a exploração de nichos através de operadores como: migração, restrições de casamento e funções de compartilhamento. É possível também utilizar informação dependente do problema para criar novos operadores que exploram conhecimento específico, visando aprimorar a busca realizada pelo Algoritmo Genético.

Parâmetros de Controle: a busca realizada pelo AG é caracterizada pela exploração de novas regiões do espaço de busca e a utilização da informação obtida sobre estas regiões para direcionar este processo (*exploration, exploitation*). O correto balanceamento entre estes aspectos é feito pela definição de parâmetros de controle. A probabilidade de recombinação P_r e a probabilidade de mutação P_m controlam a aplicação dos operadores genéticos. Aumentos no valor de P_r tendem a incrementar a combinação dos blocos de construção, incrementando, por outro lado, a destruição de boas soluções encontradas.

Incrementos no valor de P_m tendem a transformar a busca genética em uma busca aleatória, mas também propicia a reintrodução de material genético eventualmente perdido.

O tamanho da população T está associado à diversidade existente entre os indivíduos da população. Um incremento deste parâmetro provoca um aumento da diversidade de soluções, reduzindo a probabilidade de o AG convergir prematuramente para pontos ótimos locais. Por outro lado, tem-se um acréscimo do tempo computacional necessário para que ocorra a convergência para regiões ótimas do espaço (Srinivas e Patnaik, 1994).

Segundo estudos realizados por De Jong (1975), envolvendo o uso de AG em otimização de funções, um bom desempenho pode ser obtido com probabilidade de recombinação alta, probabilidade de mutação baixa, inversamente proporcional ao tamanho da população, e tamanho da população moderado. Valores típicos da ordem de grandeza desses parâmetros: $P_r = 0,6$, $P_m = 0,001$ e $T = 100$.

3.5.4. Algoritmo de Seleção Clonal

A meta-heurística *CLONALG* (CLONal selection ALGorithm) proposta por De Castro e Von Zuben é inspirada em mecanismos do Sistema Imune, mais especificamente no processo de maturação de afinidade de anticorpos, característica da resposta imune adaptativa (De Castro, 2001), (De Castro e Von Zuben, 2002) (o Apêndice C descreve sumariamente conceitos de sistemas imunes). O CLONALG tem duas versões: uma voltada ao aprendizado de máquina e uma voltada à otimização.

A versão voltada à otimização recebe as entradas: comprimento dos anticorpos (L); população Ab , formada por N anticorpos; quantidade de gerações até a parada (gen); número de anticorpos a serem selecionados para clonagem (n , sendo que nesta versão de otimização $n = N$); fator multiplicativo β , para definir quantidade de clones, e quantidade d de anticorpos com baixa afinidade que serão substituídos. Como saída, o CLONALG fornece a Matriz Ab , correspondente aos anticorpos de memória; e os valores de afinidade (fitness) f de cada anticorpo da matriz.

O CLONALG segue os seguintes passos, em cada geração (iteração):

1. Em um instante de tempo t , cada anticorpo Ab_i da população Ab tem avaliada a sua afinidade (fitness) em relação à função objetivo a ser otimizada $g()$.
2. Um vetor de afinidades f_j em relação aos anticorpos da população Ab é determinado;
3. Do conjunto Ab , um subconjunto Ab_n , composto pelos n anticorpos com maiores afinidades é selecionado, baseado na afinidade f_i de cada anticorpo Ab_i .
4. Os n indivíduos selecionados irão se proliferar em um processo de clonagem, gerando uma população de C_j clones. Quanto maior a afinidade, maior o número de clones de cada um dos n anticorpos;
5. A população de clones C_j é submetida a um processo de maturação de afinidade, gerando uma nova população C_j^* . Neste processo, cada anticorpo sofre mutação com taxa inversamente proporcional à sua afinidade.
6. É determinada a afinidade f_j^* entre o conjunto C_j^* de clones mutados.
7. Desta população C_j^* , considere todos os anticorpos como o conjunto de memória Ab_m .
8. Substitua d anticorpos de Ab_r por Ab_d novos indivíduos. Os anticorpos com menores afinidades são escolhidos para serem substituídos.

3.5.5. Otimização por Enxame de Partículas

A meta-heurística *Otimização baseada em Enxame de Partículas* – OEP (*Particle Swarm Optimization*), definida inicialmente por Kennedy e Eberhart, é inspirada na simulação de modelos sociais (Kennedy e Eberhart, 1995), (Kennedy e Eberhart, 2001). Conforme destacam os autores, a coreografia das revoadas de pássaros desperta interesse de quais são as regras subjacentes que permitem pássaros voarem próximos uns dos outros, de maneira tão sincronizada, com mudanças repentinas de direção, dispersão e reorganização. Análise semelhante pode ser feita sobre cardumes de peixes: “os membros individuais de um cardume de peixes teoricamente podem lucrar das descobertas e experiências prévias de

todos os outros membros do cardume durante a busca por alimentos”. A hipótese deste tipo de algoritmo é, portanto, que o compartilhamento de informação entre os membros do grupo oferece uma vantagem evolutiva.

Eberhart e Shi (2001) descrevem aplicações para a OEP. Cada solução potencial é representada por uma partícula, que possui uma velocidade e uma posição no espaço de busca do problema. Cada partícula mantém o melhor valor objetivo (*fitness*) e a respectiva posição alcançados até o momento corrente, valor referido como *pbest*. Também é mantido o melhor valor objetivo global *gbest* e sua posição, considerando valores de todas as posições e partículas alcançados até o momento corrente. Algumas variações não consideram valores *gbest*, ou fazem com que as partículas considerem as melhores avaliações apenas nas suas vizinhanças (ao invés de globais).

Uma meta-heurística OEP segue tipicamente os seguintes passos:

1. Iniciar uma população de partículas com posições e velocidades aleatórias nas d dimensões do espaço de busca.
2. Para cada partícula, avaliar a função objetivo (*fitness*) considerando as d variáveis.
3. Comparar o valor de objetivo da partícula com o valor *pbest*. Se o valor corrente é melhor do que *pbest*, então o valor corrente se torna o novo *pbest* e a localização corrente é registrada como localização *pbest* no espaço d -dimensional.
4. Comparar o valor objetivo da partícula com o melhor valor global corrente. Se o valor corrente é melhor do que *gbest*, então o valor corrente se torna o novo *gbest* e a localização corrente é registrada como localização *gbest* no espaço d -dimensional.
5. Mudar a velocidade e posição das partículas. Tipicamente, as partículas possuem uma massa que limita movimento. Novas posições são calculadas baseadas nas posições correntes, em componentes aleatórios, e em *pbest* e *gbest*.

6. Voltar ao passo 2, enquanto o critério de parada não for alcançado.

3.5.6. Outras Meta-heurísticas

A descrição exaustiva de meta-heurísticas está além do escopo deste trabalho. Algumas outras abordagens de otimização, também potencialmente aplicáveis em problemas de engenharia de software são: *Otimização Baseada em Colônias de Formigas – Ant Colony Optimization* (Dorigo e Di Caro, 1999); *Busca Dispersa – Scatter Search* (Glover, Laguna e Martí, 2000); e *Otimização Extrema – Extremal Optimization* (Boettcher e Percus, 1999).

3.6. Geração de Dados de Teste Baseada em Busca

Harman et al. (2009) identificaram que 59% dos trabalhos em SBSE abordam o teste de software. Os autores apresentam uma classificação desses trabalhos em relação à abordagem, tipo e nível de teste. McMinn (2004) apresenta um *survey* sobre a geração de dados de teste baseada em buscas – *Search-based Software Test Data Generation – SBST*⁴. Ali et al. (2010) apresentam uma revisão sistemática sobre a investigação empírica em trabalhos que abordam a geração de dados de teste usando técnicas de busca.

A ideia geral desses trabalhos é aplicar meta-heurística para buscar, no espaço de possíveis entradas do programa, dados de teste que satisfaçam um determinado critério. As meta-heurísticas funcionam como ferramentas para obter dados de teste que exercitem os elementos requeridos pelos critérios de teste, ou que avaliem certas características do software. O problema da geração de dados de teste é reformulado então como um problema de otimização, ou seja, de busca por um conjunto “ótimo” de dados de teste, pertencentes ao espaço de todos os possíveis dados de teste (o domínio de entrada do programa).

⁴ Dependendo do contexto SBST pode referir-se a “Search Based Software Testing”, o uso de técnicas de busca em problemas de teste de software, não necessariamente para a geração de dados de teste.

Meta-heurísticas, como Recozimento Simulado, Algoritmos Genéticos, e outras, são aplicadas em diversos cenários, tais como o teste baseado em estrutura, o teste baseado em defeitos, e o teste baseado em modelos, com ênfase em modelos de estados.

A Figura 18, adaptada de Harman et al (2009), mostra, em uma perspectiva ampla, uma abordagem SBST. Um engenheiro de software define um critério de adequação, baseado no objetivo do teste. O mesmo engenheiro define uma função objetivo baseada nos elementos (ou características) do software (ou do modelo) a serem exercitados. Uma meta-heurística usa esta função para direcionar a busca de dados de teste. A execução do software ou de modelos pode ser realizada para permitir a avaliação dos dados de teste.

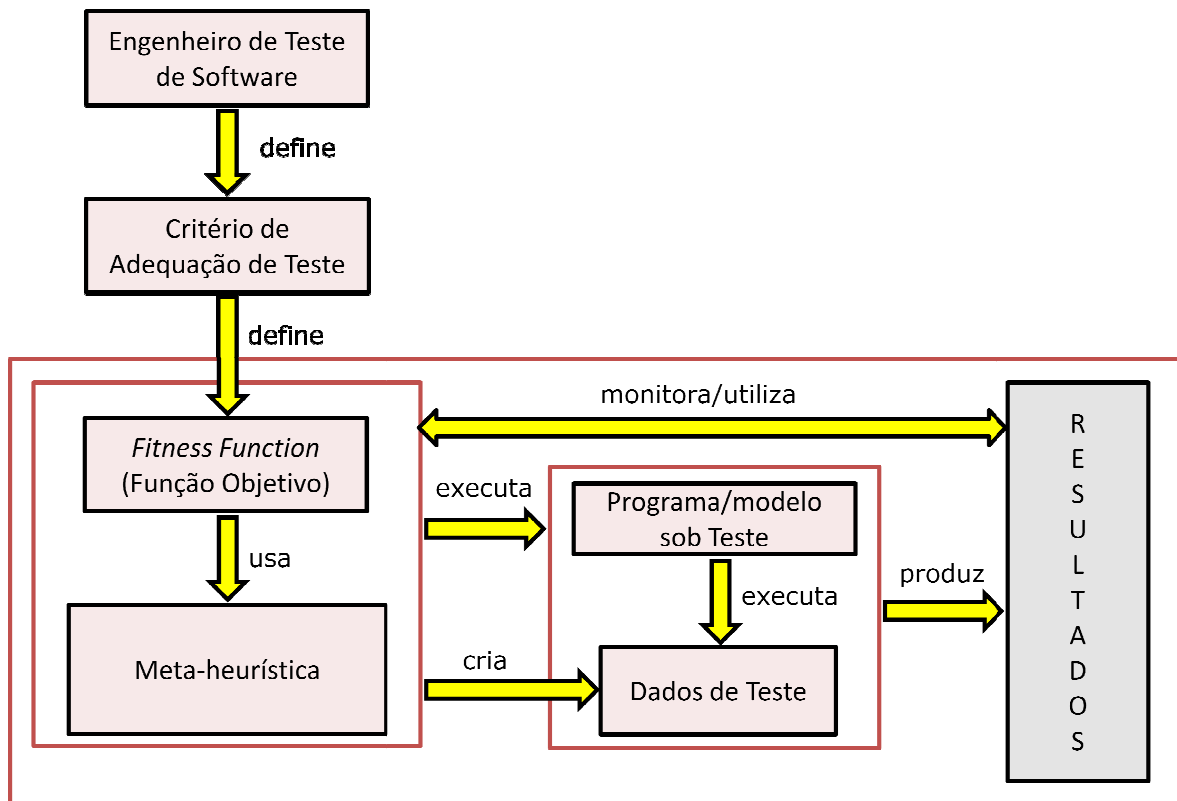


Figura 18. Elementos de uma solução de geração de dados de teste baseada em buscas

A seguir são descritos trabalhos em SBST com ênfase no teste baseado na estrutura da implementação e também no teste baseado em modelos.

3.7. Teste Baseado em Busca com Ênfase na Implementação

Uma linha importante de trabalhos em SBST visa a geração de dados de teste que exercitem a implementação do software. Esses trabalhos, descritos de forma resumida a seguir, aplicam meta-heurísticas para gerar dados de teste que exercitem comando, ramos, ou caminhos em programas.

Jones et al.

Jones et al (1996) aplicam um Algoritmo Genético com o objetivo de exercitar ramos no software.

As variáveis de entrada do programa são representadas como um string binário. Neste string as representações binárias dos valores de cada variável são concatenadas, de forma que o comprimento do string é dado pela soma dos comprimentos das representações de cada variável. Esta representação é compatível com variáveis de diferentes tipos (caracteres, inteiros, reais) e com estruturas complexas como registros.

A função de ajuste é baseada nos predicados a serem satisfeitos para o teste de ramos e direciona a busca para encontrar valores de entrada que provoquem a execução de um determinado ramo do programa. O gerador de dados de teste concentra-se em um ramo por vez, e o foco é mudado automaticamente para os próximos ramos até que todos tenham sido tratados.

Considerando o seguinte trecho:

If C1 = C2 then S1 else S2 end if.

No qual C1 e C2 são funções inteiras e S1 e S2 representam sequências de comandos distintas, executadas quando C1 = C2 é verdadeiro ou falso, respectivamente. C1 e C2 são funções, possivelmente complexas, de algumas ou de todas as variáveis de entrada inteiras {x1, x2, ..., xn}.

Valores de ajuste são calculados com base nos valores que ocorrem para C1 e para C2 imediatamente antes do comando if quando o programa é executado. Estes valores são capturados por meio da instrumentação do programa com comandos especiais. Importante notar que informações explícitas sobre o relacionamento entre as variáveis de entrada $\{x_1, x_2, \dots, x_n\}$ e C1 ou C2 não são utilizadas, apenas os valores de C1 e de C2 obtidos pela execução do programa são necessários.

Duas funções de ajuste são consideradas: 1) O inverso do valor absoluto entre C1 e C2 ($f_1=1/|C_1-C_2|$), o que faz com que valores de entrada que levem a valores de C1 próximos dos valores de C2 possuam valores de ajuste mais altos. 2) O inverso da distância Hamming entre C1 e C2 ($f_2=1/\text{Hamming}(C_1, C_2)$). A distância Hamming mapeia a diferença entre pares de strings binários para um valor numérico. O cálculo desta distância requer que os valores envolvidos no predicado (o valor de C1 e o valor C2) sejam convertidos para binário, para que a distância Hamming possa ser calculada. Para ambas as funções de ajuste f_1 e f_2 , um valor maior significa que a condição $C_1 = C_2$ está mais próxima de ser satisfeita. Um tratamento para evitar a divisão por zero no cálculo de f consiste em somar um pequeno valor positivo nos denominadores.

Como o objetivo desta abordagem é exercitar ramos e, em geral os ramos encontram-se internos na estrutura do software, as funções de ajuste devem considerar situações nas quais o ramo a ser exercitado não é alcançado na execução. Jones et. al atribuem um valor fixo baixo para o ajuste, indicando uma solução “ruim” nestas situações. Caso o ramo desejado seja alcançado, tem-se então a utilização da função baseada nos valores envolvidos nos predicados para calcular o ajuste das soluções, conforme descrito no parágrafo anterior. O fato de soluções que não atingem o ramo a ser exercitado receberem o mesmo fitness tende a criar “platôs” de valores desta função, prejudicando o direcionamento da busca do AG.

Os operadores genéticos aplicados são recombinação e mutação simples. A recombinação realiza a troca de bits entre os strings (recombinação uniforme). A seleção de soluções para a próxima geração é baseada nos valores de fitness de cada dado de teste, isto

é, quanto maior o fitness, maior a chance do dado de teste ser selecionado para a próxima geração.

Michel et al.

Michel et al (2001) definem o problema da geração de dados de teste como “encontrar dados de entrada que satisfaçam um critério de adequação definido”. Os autores destacam uma ideia essencial da geração dinâmica de dados de teste: se um determinado requisito de teste não é exercitado (satisfeito), os dados coletados durante a execução podem ainda ser utilizados para determinar quais dados estão mais próximos de satisfazer o requisito de teste.

Uma ferramenta chamada GADGET (Genetic Algorithm Data GEneration Tool) implementa a abordagem para a geração de dados proposta. A geração de dados é baseada em um Algoritmo Genético, que tem como objetivo encontrar conjuntos de dados de teste que satisfaçam o critério Cobertura de Condições e de Decisões. São gerados, dois requisitos de teste para cada condição: que a condição seja verdadeira pelo menos uma vez, e que a condição seja falsa pelo menos uma vez.

Antes de iniciar a busca com o Algoritmo Genético (AG) o programa é executado com uma entrada semente. Este dado de teste semente executa algum caminho no software e tipicamente exercita vários requisitos de teste. Uma tabela de cobertura é criada para registrar, para cada condição do programa, se ela foi exercitada com valor verdadeiro e também com valor falso pelo menos uma vez durante esta primeira execução.

Esta tabela de cobertura é utilizada para selecionar os requisitos de teste (condições), um de cada vez, para a geração de dados de teste. Para cada novo requisito a ser exercitado, o AG é utilizado para buscar dados de teste que exercitem o requisito. Se nenhum dos ramos de uma determinada condição foi exercitado, isto significa que esta condição não foi atingida até então e, portanto, não é possível tratá-la ainda. Por outro lado, se apenas um ramo da condição foi exercitado, isto indica que a condição já foi atingida por algum dado de teste, mas um dos ramos da condição ainda precisa ainda ser exercitado. Neste caso, o requisito de teste (ramo da condição) é selecionado para a próxima busca com o AG. A

população inicial do AG é formada por dados de teste que atingiram esta condição (embora não a tenham avaliado conforme o requisito atual) e dados gerados aleatoriamente.

A tabela de cobertura é atualizada a cada novo requisito exercitado, o que pode ocorrer quando o requisito é o alvo da busca, ou quando o requisito é exercitado “por sorte” durante a busca para exercitar outro requisito.

Supondo que um programa hipotético tenha o comando condicional `if (pos >=21) ...` na linha 324 e que o objetivo é garantir que o ramo “verdadeiro” seja seguido. É necessário encontrar um dado de entrada que faça com que a variável *pos* tenha um valor maior do que ou igual a 21 quando a linha 324 é alcançada.

Considerando $pos_{324}(x)$ o valor de *pos* guardado na linha 324 quando o programa é executado com a entrada *x* (notar que *x* representa as variáveis de entrada do programa, isto é, um dado de teste). Então a função:

$$E(x) = 21 - pos_{324}(x), \text{ se } pos_{324}(x) < 21;$$

$$E(x) = 0 \text{ caso contrário.}$$

É mínima quando o ramo verdadeiro é seguido na linha 324.

Portanto, o problema da geração de dados de teste é reduzido a um problema de minimização de funções: para encontrar o dado de teste desejado, é necessário encontrar um valor de *x* que minimize $E(x)$. A função $E(x)$ é chamada de função objetivo (ou *fitness*) e informa ao gerador de dados de teste o quão próximo ele está de alcançar o seu objetivo. O AG representa as variáveis de entrada como um string de bits e utiliza o operador de recombinação uniforme e a mutação simples.

Pargas et al.

Pargas et al. (1999) apresentam uma técnica orientada a objetivos para a geração automática de dados de teste. Assim como nos trabalhos anteriormente descritos, um AG é utilizado para a geração de dados. O Grafo de Dependências de Controle do programa em teste (GDC) é utilizado para direcionar a busca do AG. Este grafo é definido a partir do Grafo de Fluxo de Controle do programa (GFC) e explicita relações de pós-dominância

entre nós existentes neste grafo. Essencialmente, um caminho acíclico no GDC, a partir do nó de início do grafo até um nó N , contém um conjunto de predicados que precisam ser satisfeitos por um dado de teste para que N seja executado.

O AG utilizado representa como indivíduos da população dados de teste. A codificação das variáveis de entrada é na forma de inteiros concatenados para formar um dado de teste.

A abordagem da geração de dados define os seguintes passos:

Pré-processamento: o GDC é formado e caminhos são identificados. Para cada comando do programa a ferramenta (chamada GenerateData) gera caminhos acíclicos no GDC que contém os predicados que ficam entre o comando e o nó de entrada do GDC. Os requisitos de teste são identificados e é criado um “scoreboard” para o registro dos requisitos de teste exercitados. A população inicial do AG é gerada aleatoriamente.

Geração de dados: a busca do AG ocorre até que todos os requisitos sejam executados. O AG focaliza um requisito por vez, denominado *Target*. O fitness de cada indivíduo (dado de teste) é calculado com base nos caminhos no GDC. Se um dado de teste t_i executa um caminho no programa que contém mais predicados no GDC para o *Target* comparado ao caminho executado por um dado de teste t_j , então t_i recebe um valor de fitness mais alto do que T_j . A lógica é que um dado de teste está “mais próximo” de exercitar o target se ele provoca a execução de um caminho que tem um número alto de predicados em comum com o caminho no GDC deste *Target*.

A seleção de indivíduos aloca uma chance de seleção proporcional ao fitness do indivíduo. São aplicados os operadores genéticos de recombinação de ponto único e mutação, que consiste em trocar o valor de uma variável de entrada para um valor gerado aleatoriamente. Cada indivíduo da população é executado no programa em teste. Se pelo menos um indivíduo executa o *Target*, a GenerateData passa a focalizar automaticamente no *Target* seguinte, eventualmente a busca é encerrada se um número máximo de tentativas sem sucesso para executar um *Target* for alcançado.

Importante destacar que a função de ajuste adotada para avaliar os dados de teste considera o número de predicados resolvidos corretamente em direção ao *Target*, mas não faz distinção entre as soluções que executam um mesmo número de predicados. Como consequência, a busca para resolver os predicados não tem um direcionamento adequado.

Wegener et al.

Wegener et al (2002) abordam o uso de meta-heurísticas para o teste de software embarcado. Em relação ao teste estrutural a abordagem consiste em representar um dado de teste como um indivíduo da população. O objetivo é gerar dados de teste que levem a melhor cobertura possível de critérios estruturais Todos os Comandos e Todos os Ramos.

Semelhante à abordagem de Michel et al (2001) esta proposta considera cada requisito (nó ou ramo) por vez. Otimizações independentes usando o algoritmo evolutivo são realizadas para cada requisito (também referido como “objetivo parcial”). É monitorado quando algum requisito não alvo da busca é exercitado ao acaso. São também guardados dados de teste próximos de satisfazer requisitos não alvo, usados para a formação da população inicial em otimizações posteriores.

A função de ajuste utilizada é composta por dois componentes: o nível de aproximação e a distância local. O nível de aproximação de um determinado indivíduo (dado de teste) é igual ao numero de nós com predicados (*branching nodes*) que se encontram entre os nós exercitados pela execução do indivíduo e o nó desejado (alvo da busca). A *distância local* consiste nas condições de ramo nas quais o desvio em relação ao nó desejado ocorre. Por exemplo, considerando o objetivo de avaliar como verdadeira a condição ($x==y$) em um ramo, a função objetivo é $|x-y|$. Mais precisamente, e utilizando a nomenclatura dos autores, define-se:

$$F(pa,i)=AL(pa,i)+(1-LD(N(pa, AL(pa,i)),i)$$

Na qual,

$F(pa,i)$: função objetivo do indivíduo i em relação ao objetivo parcial pa ;

$AL(pa,i)$: maior nível de aproximação para o objetivo parcial pa ;

$N(pa, al)$: nó predicado com o maior nível de aproximação para o objetivo parcial pa ;

$LD(n, i)$: distância normalizada para o indivíduo i no nó predicado n .

Apesar do componente distância local da função objetivo ser o mesmo que a função de predicado nos trabalhos anteriores, esta abordagem significa um aprimoramento por considerar o nível de aproximação. Este componente distingue os indivíduos que, por qualquer caminho, chegam mais próximos do nó objetivo.

Windisch et al.

Windisch et al. (2007) utilizam Otimização baseada em enxame de partículas para o teste estrutural. O algoritmo utilizado é chamado de Otimizador de Partículas de Aprendizado Abrangente (CL-PSO) (Liang et al., 2006).

O algoritmo inicia com uma população de dados de teste gerados aleatoriamente. Cada dado de teste é uma partícula que possui uma posição e uma velocidade correntes. Cada partícula mantém a melhor posição explorada por ela até o momento, e o enxame como um todo também mantém a melhor solução encontrada por todos os membros até o momento.

Durante o processo de busca as partículas exploram o espaço d-dimensional com trajetórias que são influenciadas pela trajetória de cada partícula e também pelas partículas vizinhas no enxame (d é o numero de parâmetros de entrada do programa). O algoritmo CL-PSO aplica uma estratégia de aprendizado em que cada partícula aprende de diferentes vizinhos para cada dimensão separadamente. Uma taxa de aprendizado P_c determina o quanto uma partícula apreende dos melhores valores de objetivo alcançados por ela e por outras partículas em cada dimensão. Vetores com d dimensões são utilizados para manter a posição e a velocidade de cada partícula. São mantidas também as melhores posições visitadas por cada partícula com respeito a cada dimensão. Tanto a posição quanto a velocidade das partículas situam-se dentro de limites estabelecidos.

Para cada requisito a ser exercitado (ramo ou comando do programa) uma busca independente é realizada pelo algoritmo CL-PSO, que procura pelo dado de teste que

exercite o requisito. Embora não explicitado no artigo, a função objetivo utilizada parece ser baseada na distância para satisfazer os ramos, semelhante à de (Wegener et al., 2002).

Abreu et al.

Abreu, Martins e Souza (2007) abordam a geração de dados de teste para executar caminhos em programas. É utilizada a metaheurística Otimização Generalizada Extrema (OGE) para a geração de dados, uma variação da Otimização Extrema (Boettcher e Percus, 1999).

OGE tem natureza evolutiva e difere dos Algoritmos Genéticos por associar valores de objetivo (fitness) para partes de um indivíduo (chamadas de espécies) ao invés de associá-los ao indivíduo. A população é composta por um único string indivíduo que codifica de forma concatenada as variáveis representadas em binário.

O primeiro passo do algoritmo OGE consiste em gerar aleatoriamente um string binário com m bits. É feita uma mutação em cada bit do string, um bit por vez, e a variação do valor objetivo (fitness) dv é calculado. Os bits são listados em um ranque de acordo com os valores dv obtidos, de valores menores de dv para valores maiores – considerado uma busca para maximizar a função objetivo. A probabilidade de um bit sofrer mutação é influenciada pela posição do bit no ranque de variação da função objetivo (dv). A lógica é que mutações de bits que acarretam uma melhora expressiva na função objetivo são favorecidas.

A função objetivo utilizada NEHD (Normalized Extended Hamming Distance) foi definida por Lin e Yeh e compara dois caminhos de programas retornando um valor de similaridade entre eles (Lin e Yeh, 2001). Deste modo, dados de teste que executam caminhos mais próximos do caminho desejado (segundo a medida NEHD) recebem um valor maior para a função objetivo. Uma breve descrição de como a medida NEHD é calculada é fornecida em (Abreu, Martins e Souza, 2007).

Bueno e Jino

Bueno e Jino (1999) propõem a aplicação de um Algoritmo Genético Simples (AG) (Goldberg, 1989) para o problema da geração de dados de teste para exercitar caminhos completos em programas (Bueno, 1999), (Bueno e Jino, 2000), (Bueno e Jino, 2001), (Bueno e Jino, 2002).

A abordagem tem como características principais:

- A manutenção de uma “base de soluções”, incrementada durante o processo de busca, e que contém informações sobre dados de entrada e respectivos caminhos executados. Esta base serve para que o AG inicie a busca de dados para executar um caminho objetivo com uma população inicial de dados de teste que executaram caminhos similares ao objetivo.
- É realizado um monitoramento do progresso da busca feita pelo AG para executar um dado caminho. Em caso de ausência persistente do progresso na busca é emitida uma mensagem de possível não executabilidade do caminho objetivo. O número e o tipo de predicados são informações utilizadas em uma heurística para ajustar o monitoramento, tolerando uma maior ausência de progresso (sem advertência) para caminhos cuja geração de dados é mais difícil.

O AG representa as variáveis de entrada do programa em forma binária e concatenada em um string. O testador deve fornecer faixas de valores para variáveis do tipo numérico. Para variáveis do tipo real, deve fornecer adicionalmente a precisão desejada. Caracteres são representados como inteiros e strings (cadeias de caracteres) são tratadas como variáveis independentes do tipo caractere. O tratamento de programas que aceitam outras estruturas é feita por meio de drivers que traduzem o formato de variáveis do AG para estruturas aceitas pelo programa.

A cada geração do AG são aplicados os operadores genéticos de recombinação simples e de mutação de ponto único e é feita a seleção de indivíduos (dados de teste) proporcional aos seus valores de ajuste, definidos por meio da função:

$$F_t = NC - (EP/MEP)$$

Na qual NC é o número de nós coincidentes do caminho executado em relação ao caminho objetivo, contados do nó de entrada do programa ao nó em que ocorre o desvio do fluxo em relação ao caminho objetivo.

EP é o valor absoluto da função de predicado associada ao ramo em que houve o desvio do caminho desejado. Representa o erro que provocou o desvio do caminho executado em relação ao objetivo e trata-se da função de predicado definida por Korel (1990). Por exemplo, para o ramo verdadeiro no comando $if(y == 100)$ $EP = |y - 100|$.

MEP é o valor máximo para a função predicado entre todos dados de teste que executaram corretamente o mesmo número de nós em relação ao caminho desejado, isto é desviaram do caminho objetivo no mesmo predicado.

Esta função objetivo reflete o fato de que quanto maior o número de nós executados corretamente em relação ao caminho desejado, mais próxima a solução está do dado de teste desejado. Considerando que vários dados de teste executam o mesmo número de nós corretamente, os melhores são aqueles que apresentam um menor valor absoluto para a função de predicado associada ao ramo onde ocorre o desvio em relação ao caminho desejado. Esta função objetivo é compatível com caminhos que exercitem iterações em laços, sendo assim aplicável para exercitar componentes estruturais mais complexos, como associações de fluxo de dados.

Tonella

Tonella (2004) aborda o teste de programas orientados a objetos. Os casos de teste para métodos de classes incluem a criação de um ou mais objetos, as mudanças de estados internos e a invocação do método em teste. O trabalho focaliza a representação dos casos de teste na forma de indivíduos para a utilização de técnicas de computação evolutiva. Além de representar valores de entrada passados como parâmetro, os indivíduos representam também sequências de comandos.

O procedimento do teste de classes possui os seguintes passos:

1. Um objeto do componente em teste é criado usando um construtor;

2. Uma sequencia de zero ou mais métodos é invocada neste objeto para levá-lo a um estado apropriado;
3. O método em teste é invocado;
4. O estado final alcançado pelo objeto é examinado para avaliar o resultado do caso de teste.

O AG utilizado para a geração de dados é baseado no definido em Pargas et al (1999). O conjunto de elementos a serem exercitados (e.g., ramos) é definido e buscas com o AG são realizadas para cada elemento alvo (que se pretende exercitar), um por vez. A função objetivo de cada dado de teste (indivíduo do AG) é calculada considerando grafos de dependência de controle. Mais especificamente o valor da função objetivo considera a proporção de ramos executados em relação a todos os ramos que levam ao ramo alvo, trata-se, portanto, de uma função semelhante à de (Pargas et. al., 1999).

A sintaxe dos indivíduos é definida por meio de uma gramática e determina duas partes. A parte definida pelo não terminal <actions> contém uma sequência de construtores e invocações de métodos. A segunda parte contém os valores de entrada a serem usados nas invocações de métodos. Os valores de entrada podem ser de tipos básicos (inteiro, real, lógico) e são gerados aleatoriamente com valores entre um limite mínimo e um limite máximo. Valores podem também serem gerados por meio de uma classe externa, que considera regras específicas para a geração de dados.

Operadores de mutação definidos realizam basicamente troca de variáveis de entrada; mudanças em construtores; inserção ou remoção de invocações de métodos.

3.8. Teste Baseado em Buscas com Ênfase em Modelos e/ou na Especificação

Conforme descrito no Capítulo 2, o teste de software baseado em modelos baseia as tarefas de teste, como a geração de dados de teste e a avaliação de resultados, em um modelo do software em teste. Os trabalhos descritos a seguir aplicam meta-heurísticas para gerar dados de teste que exercitem estruturas presentes em modelos.

Derderian et al.

Derderian et al. (2005) abordam o teste de sistemas modelados como máquinas de estados finitos estendida (EFSM). São consideradas EFSMs que possuem parâmetros de entrada e de saída, variáveis internas, operações e predicados definidos com essas variáveis e com os parâmetros de entrada. Rótulos de cada transição t possuem duas guardas que determinam a quando t é executada. Uma guarda refere-se a entradas (G_i) e outra se refere a domínio (G_d). A execução de um caminho de transições na EFSM depende dos resultados das avaliações dessas guardas (G_i e G_d), que são determinados dinamicamente dependendo dos valores das variáveis internas e das entradas. Cada entrada, por sua vez, pode assumir diferentes valores após cada transição.

O trabalho aborda a geração de caminhos de transição executáveis (FTP), de um estado si para um estado sj e a geração de uma sequência de entradas necessária para que os caminhos sejam executados. Transições da EFSM são categorizadas em tipos, relacionados ao nível de dificuldade para serem satisfeitas, baseado na presença de ou não de guardas tipos G_i e G_d . A lógica é que a presença em um caminho de transições simples, com poucas condições de guarda, aumenta a chance de que o caminho seja executável (seja um FTP). Uma função objetivo estima a probabilidade de um caminho de transições ser executável, e é utilizada por um Algoritmo Genético para a geração de dados.

Kalaji et al.

Kalaji et al. (2009) também abordam a geração de dados para máquinas de estados finitos estendidas (EFSM).

A função de ajuste estima a chance de um caminho na EFSM ser executável. Esta classificação é feita pela categorização de transições na máquina de estados em “que afetam” e “afetadas”. Transições “que afetam” têm uma operação que pode afetar guardas de uma transição seguinte, a “afetada”. Cada par de transições (que afeta, afetada) é associada a um valor numérico dependendo do tipo de atribuições e guardas encontrados no par. Este valor de função objetivo representa o quão fácil é satisfazer a guarda, sendo que um valor menor de objetivo indica que é mais provável que o caminho seja executável.

O problema da geração de dados para um caminho na EFSM é considerando como determinar uma sequência de entradas fornecidas para levar à execução das transições desejadas. A função objetivo proposta é baseada na de Wegener et al. et al (2002) que conjuga dois componentes. Com relação ao caminho desejado uma *função de aproximação* reflete o quão próximo à entrada está da transição a ser executada e uma *função de distância* reflete o quão próximo à transição está de ser satisfeita.

A abordagem para a geração de dados considera duas fases: inicialmente uma abordagem evolutiva é utilizada para selecionar um caminho de transições com maior probabilidade de ser executável, e em seguida outra abordagem é aplicada para tentar executar o caminho selecionado. Em caso de falha da segunda fase a primeira fase é repetida.

Yano et al.

Yano, Martins e de Souza (2010) apresentam uma abordagem para derivar sequências de teste a partir de uma EFSM usando algoritmos evolutivos. O objetivo é exercitar transições no modelo pela execução de algum caminho; para evitar a geração de caminhos não executáveis é utilizada uma versão executável instrumentada do modelo. O algoritmo empregado é multi-objetivos M-GEO (Multi-Objective Generalized Extremal Optimization), uma variação multi-objetivo do algoritmo GEO, uma variação da Otimização Extrema (Boettcher e Percus, 1999).

A abordagem para a geração de sequências segue os seguintes passos. Um módulo de análise de modelo produz informação de slice para a transição alvo. Esta informação contém todas as transições e estados que podem afetar os valores envolvidos na transição. Usando o slice do modelo são identificadas transições críticas, que são transições que causam um desvio em relação à transição alvo.

O modelo executável implementa o comportamento do modelo em uma linguagem de programação. Este modelo recebe uma sequência de teste gerada como entrada e produz um caminho de transições provocado pela sequência, deste modo é possível verificar as transições efetivamente exercitadas e garante-se que um caminho executável é produzido.

O módulo gerador de sequencias de teste usa a informação do slice e as transições efetivamente exercitadas para avaliar cada solução. O algoritmo M-GEO é utilizado para evoluir as soluções usando duas funções objetivo baseado no conceito de Ótimo de Pareto. A primeira função F1 direciona as soluções para exercitar o alvo do teste. Esta é baseada em abordagens de teste estrutural e é calculada em termos do nível de aproximação em relação á transição alvo e no erro que provoca o desvio do alvo em alguma transição crítica a segunda função F2 busca minimizar o tamanho da sequencia de teste.

Zhao et al.

Zhao et al. (2010) apresentam um sistema para a geração automática para caminhos de transições executáveis em EFSM.

Um Algoritmo Genético busca por um conjunto de dados de entrada que provoque a execução de um caminho completo na EFSM. Um indivíduo é uma lista de valores de entrada $x = (x_1, x_2, \dots, x_n)$ correspondentes a todos os parâmetros dos eventos (e_1, e_2, \dots, e_m) na ordem em que eles aparecem.

Em um primeiro passo caminhos completos com diferentes comprimentos são gerados nos modelos EFSM utilizados no experimento. Caminhos não executáveis são identificados e removidos. Em um segundo passo tem-se uma busca para a geração de dados para cada caminho separadamente. Uma população de indivíduos é gerada aleatoriamente, cada um codificando os parâmetros de entrada como números reais. A função objetivo utilizada baseia-se nos componentes, o nível de aproximação (*approach level*) e distância de ramo normalizada – *norm(d)*:

$$Fitness = approach_level + norm(d)$$

$$norm(d) = 1 - 1.001^d$$

O nível de aproximação avalia o quão próximo o indivíduo esta do caminho alvo, enquanto a distância de ramo mede o quão próximo à primeira condição não satisfeita está de ser verdadeira. Trata-se de uma função objetivo semelhante à utilizada no teste estrutural para exercitar objetivos (Wegener et al., 2002), ou caminhos (Bueno e Jino, 1999). O

procedimento de seleção ocorre de acordo com os valores da função objetivo dos indivíduos.

Zhan e Clark

Zhan e Clark (2005) descrevem um método para a geração automática de dados de teste usando um algoritmo de recozimento simulado para modelos Matlab/Simulink. Simulink é um software para modelagem, simulação e análise de sistemas dinâmicos. Recursos de simulação permitem que o modelo seja executado e observado.

São introduzidos defeitos no modelo em teste por meio de operadores de mutação. Esses defeitos provocam perturbação nos valores pela adição, multiplicação, ou atribuição de valores presentes nas entradas dos blocos. Os operadores de mutação são aplicados aos modelos e mutantes são gerados com respeito a vários pontos para a injeção de defeitos (obs: este trabalho utiliza conceitos de teste baseado em defeitos, decidiu-se, no entanto, mantê-lo nesta seção que enfatiza o uso de modelos).

São gerados dados que matam os mutantes, isto é, que fazem com que a saída final do modelo mutante seja diferente da saída do modelo original. A avaliação de quão bem um dado de entrada satisfaz um requisito de teste é baseado em o quão distante o defeito injetado se propaga no modelo mutante em relação ao original, para qualquer caminho entre o defeito introduzido e a saída do modelo.

A função objetivo é calculada pela comparação dos estados de execução do modelo original e do modelo mutante. A função objetivo considera valores detectados em condições de ramos nos comandos do tipo switch por meio da inserção de pontos de prova. São computados custos relacionados às diferenças de valores entre o modelo e o mutante no comando para compor o valor objetivo a ser minimizado para obtenção do dado de teste que mate o mutante.

Ghani et al.

Em um trabalho posterior Ghani et al (2009) comparam o desempenho de algoritmos de busca para exercitar modelos Matlab/Simulink. Além de considerar blocos switch, são tratados também blocos de operadores lógicos e blocos de operadores relacionais.

Um Algoritmo Genético baseado em um toolbox da empresa MathWorks (<http://www.mathworks.com/>) é comparado a um algoritmo de Recozimento Simulado para a geração de dados de teste. Dois experimentos foram realizados.

A análise dos resultados apresentados pelo AG e pelo RS mostra que a taxa de sucesso para a geração de dados do AG é substancialmente melhor do que o RS, sobretudo em problemas mais complexos. Em relação ao número de avaliações (tentativas) o desempenho dos dois algoritmos é próximo.

Blanco et al.

Blanco et al. (2009) apresentam uma abordagem para geração de dados de teste para serviços web especificados em BPEL – *Business Process Execution Language*. Especificações *BPEL* representam o comportamento de processos de negócio baseados em composições de serviços web. Essas especificações são documentos XML compostos por uma parte de declarações de *partnerlinks* (serviços que interagem com o processo de negócios) e de *portTypes* (detalhes de interfaces entre os serviços), além de uma parte de especificação dos processos de negócio propriamente ditos.

A técnica de busca utilizada é a Busca Dispersa – BD (Scatter Search), método evolutivo que trabalha com uma população de soluções para o problema a ser resolvido (Glover, Laguna e Martí, 2000). Soluções são mantidas em um conjunto denominado Conjunto Referência e são combinadas iterativamente na tentativa de obter soluções melhores considerando critérios de qualidade e de diversidade.

As variáveis de entrada dos modelos são variáveis recebidas pelos serviços web que interagem na especificação BPEL. Um dado de teste é definido pelos valores das variáveis de entrada e das transições executadas do processo de negócio. O processo de negócio

BPEL é representado por meio de um grafo de estados no qual os nós representam estados do processo de negócio e os arcos representam uma mudança de estados de um nó i para um nó j , que ocorre quando a decisão associada ao arco (i, j) é verdadeira. O modelo é instrumentado para a avaliação dos caminhos executados pelos dados de teste.

O objetivo do sistema é gerar dados de teste que façam com que cada transição dos processos de negócio seja exercitada. Este objetivo geral é dividido em sub-objetivos, cada um consiste em gerar dados que alcancem uma transição particular T_k do grafo de estados.

Em cada iteração o TCSS-LS seleciona uma transição alvo e gera o conjunto de soluções para o Conjunto Referência da transição. A solução final fornecida são os casos de teste que exercitam cada transição do modelo.

3.9. Outras Abordagens de Teste Baseado em Busca

A geração de dados multi-objetivo é apresentada por Lakhotia et al (2007). A abordagem visa a gerar dados de teste para exercitar ramos em programas. Além deste objetivo, busca-se também exercitar a alocação dinâmica de memória. É empregada a abordagem de otimização multi-objetivo de Pareto com a utilização de um Algoritmo Genético. Pinto e Vergilio (2010) seguem uma abordagem semelhante, também empregando o conceito de dominância de Pareto, com os objetivos de: exercitar programas orientados a objetos com uso de critérios de teste estrutural; otimizar o tempo de execução dos dados de teste; e otimizar a habilidade de revelar defeitos considerando o critério análise de mutantes.

Afzal et al (2009) apresentam uma revisão sistemática de trabalhos que realizam o teste baseado em busca (SBST) de propriedades não funcionais de sistemas. As principais áreas de aplicação identificadas são:

Teste de requisitos de tempo real em sistemas embarcados, o que envolve encontrar piores casos (e melhores casos) de tempos de execução e determinar se restrições de temporização são satisfeitas. Exemplos desses trabalhos: (Weneger et al., 1997), (Alander et al., 1997), (Briand et al., 2005).

Teste de qualidade de serviços, em que meta-heurísticas são aplicadas para direcionar a composição de componentes e também para violar acordos de nível de serviço (Service Level Agreements). Exemplos: Canfora et al (2005) e Di Penta et al (2007).

Detecção de vulnerabilidades de segurança, como a detecção de sobrecarga de armazenamento (buffer overflow), realizada por meio de meta-heurísticas que buscam por atribuições de dados propensas a gerar violações (Kayacik, 2005). Ou modelar o comportamento de hackers evoluindo scripts de comandos UNIX com uso de algoritmos genéticos (Budynek et al., 2005).

Tracey et al. (1998) propõem um framework genérico para a geração de dados de teste baseada em otimização, permitindo a automação da geração para critérios de teste estrutural, para critérios de teste funcional, e também para o teste de propriedades não funcionais. O teste de falhas em relação a uma especificação é tratado com o uso de uma especificação formal com o uso de notação SPARK-Ada, que define expressões para prova que consistem de pré-condições e de pós-condições para um subprograma. Os autores abordam também o teste de condições de exceção, com foco em situações em que a condição acontece, por exemplo, a ocorrência de um problema de *overflow* numérico.

3.10. Pontos-chave e um Procedimento Genérico para Abordagens SBST

Embora os trabalhos citados neste capítulo detalhem o problema de geração de dados tratado, a meta-heurística empregada e outros detalhes do problema particular, não foi identificada em nenhum trabalho uma descrição genérica de como aplicar SBST tendo em vista um problema de geração de dados de teste. Harman et al. (2009) aproximam-se desta proposta, mas além de o escopo ser mais abrangente (SBSE e não SBST), a ênfase da análise está em classificar as abordagens (SBST para o teste estrutural, teste baseado em modelos, teste de mutantes, teste de aspectos temporais, teste de exceções, etc.) e não em identificar aspectos comuns das abordagens. O trabalho de (McMinn, 2004) aborda detalhadamente a aplicação de SBST no teste estrutural e identifica direções futuras, mas também não discorre sobre semelhanças e pontos em comum das abordagens.

Neste sentido, duas contribuições secundárias desta tese foram desenvolvidas e apresentadas a seguir:

- A análise dos vários trabalhos de SBST permitiu identificar problemas comuns a serem resolvidos, independente do objetivo de teste da abordagem (ex: exercitar, código, exercitar modelos, ou testar aspectos não funcionais). Esses problemas comuns foram organizados como pontos-chave em um guia de sete passos genéricos a serem realizados para a concepção e definição de uma nova abordagem SBST.
- Em um nível de abstração mais baixo, esta mesma análise dos vários trabalhos de SBST permitiu representar, por meio de seis passos de processamento, um procedimento genérico e simples para a geração de dados com o uso de meta-heurísticas.

Pontos-chave para abordagens SBST

Foram identificados pontos-chave na definição de uma abordagem SBST. A partir desses pontos, propõem-se os seguintes passos genéricos a serem realizados para a concepção e definição de uma nova abordagem SBST:

1. **Definir o elemento a ser exercitado no teste**, tal como ramos ou caminhos em programas, ou transições em modelos de estados. Há duas possíveis formas de empregar busca:
 - A meta-heurística pode ser empregada para realizar várias buscas, cada uma com o objetivo de encontrar um dado de entrada que exercite um elemento.
 - A meta-heurística pode ser empregada uma única vez, com o objetivo de encontrar um conjunto de dados de teste que exercite (como um todo) os vários elementos.

É possível também que o objetivo seja avaliar alguma característica (exemplo: tempo máximo de execução), ou ainda satisfazer múltiplos objetivos simultaneamente, eventualmente objetivos conflitantes.

2. **Selecionar uma meta-heurística.** Selecionar uma meta-heurística, como Subida de Encosta, Algoritmo Genético, Recozimento Simulado, etc.
3. **Definir mecanismos de variação.** Tais mecanismos permitem a exploração do espaço de busca, realizada por meio de variações a cada iteração (ex: recombinação, movimento, mutação). Normalmente, esses mecanismos estão relacionados às meta-heurísticas.
4. **Definir uma representação para as soluções.** As soluções podem representar dados de teste. Neste caso, valores de entradas podem ser codificados como variáveis do tipo real, tipo inteiro, ou usando codificação binária. As soluções podem também representar conjuntos de dados de teste. Neste segundo caso, os vários dados de teste (e os valores associados) podem ser concatenados em um único string.
5. **Definir uma função objetivo (*fitness*).** Esta função deve quantificar, tão precisamente quanto possível, o quão próxima uma solução candidata (dado de teste, ou conjunto de dados de teste) está de alcançar o objetivo estabelecido no teste. Valores da função objetivo associados a cada solução candidata permitem o direcionamento da busca visando maximizar o objetivo.
6. **Definir um modelo de instrumentação.** Para definir os valores da função objetivo para cada solução candidata, são necessárias informações sobre a execução de cada solução candidata. Esta informação pode estar associada ao código executado ou a um modelo executado. Em ambos os casos, é necessário definir comandos especiais para capturar informações da execução.
7. **Definir um procedimento de avaliação da abordagem.** Ao se definir uma abordagem para a geração de dados de teste, é necessário realizar procedimentos de avaliação da abordagem. Questões relacionadas à eficácia e à eficiência para a geração de dados podem ser avaliadas comparando a abordagem proposta com outras existentes. Comumente, a geração aleatória de dados é utilizada como base de comparação. Quando há aspectos de aleatoriedade envolvidos, várias instâncias

devem ser obtidas e analisadas estatisticamente (Barr et al., 1995), (Shull et al., 2008), (Ali et al., 2010). Por exemplo, os valores de cobertura estrutural alcançados por conjuntos de teste gerados aleatoriamente, apresentam variações de um conjunto para outro. Neste caso, são necessárias várias instâncias de conjuntos de teste para a avaliação.

Procedimento Genérico para Abordagens SBST

A Figura 19, a seguir, ilustra um procedimento genérico para a geração de dados de teste. Dependendo da meta-heurística específica, porém, alguns passos extras podem existir, ou ainda passos podem ser suprimidos. Os passos são descritos a seguir:

1. Um gerador aleatório gera um conjunto de N soluções candidatas – $SC \{sc_1, sc_2, \dots, sc_N\}$. Cada sc_i representa um dado de teste, ou um conjunto de dados de teste. Dependendo da meta-heurística utilizada, $N=1$, ou $N>1$.
2. O software ou modelo base para a geração de dados é executado utilizando as soluções candidatas sc_i de SC . A instrumentação do software ou modelo captura informações sobre a execução de cada sc_i .
3. Com base na informação da execução, cada uma das soluções sc_i é avaliada pela aplicação da função objetivo, resultando em valores objetivo f_i .
4. N soluções candidatas são selecionadas e copiadas para um novo conjunto SC' .
5. SC é submetido a um processo de variação, gerando um novo conjunto SC'' .

Cada sc_i em SC'' é executado no software ou modelo, ou seja, retorna-se ao passo 2.

Os passos 2 a 5 são executados até que um critério de parada seja alcançado. Tipicamente, o critério de parada é alcançar o objetivo do teste, por exemplo, exercitar os requisitos de teste (ramos, caminhos, etc.) – o que é representado no passo 6, ou exceder um limite superior de iterações.

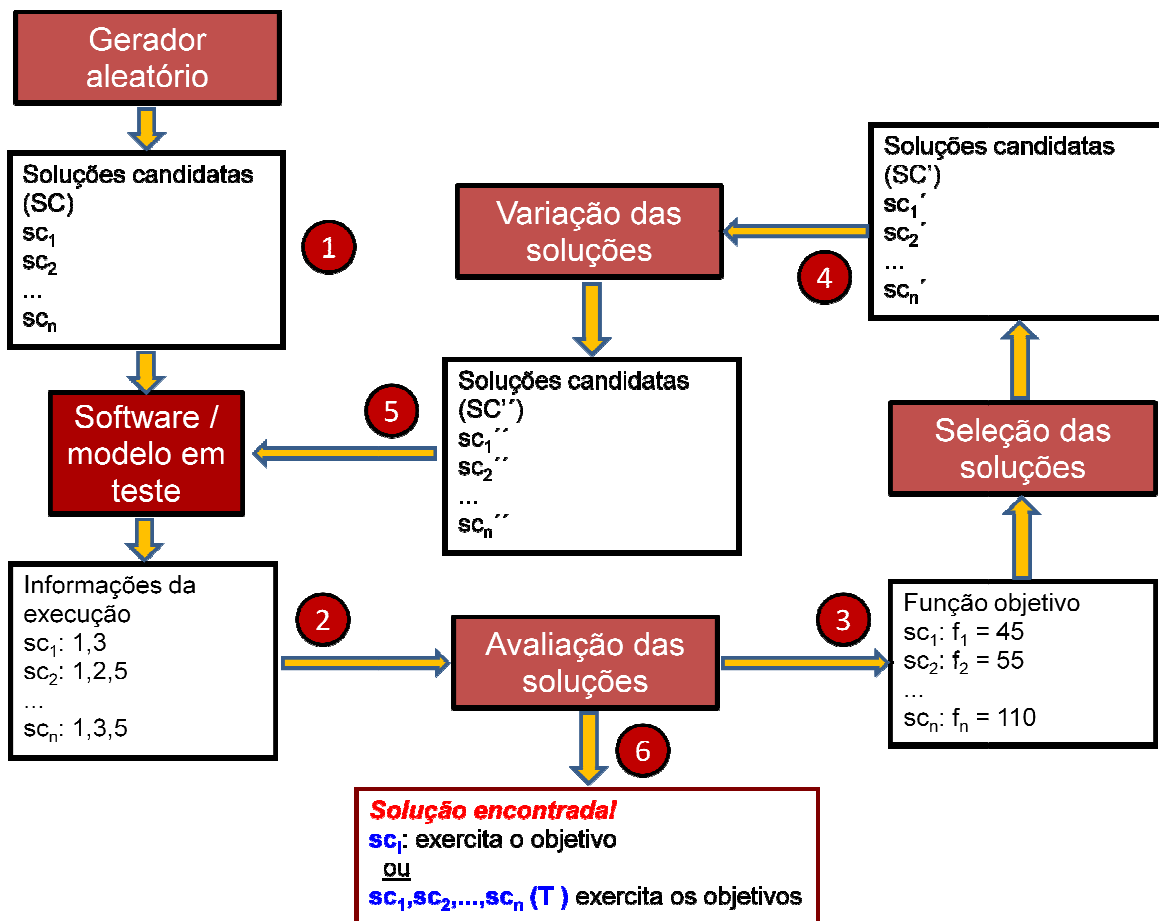


Figura 19. Procedimento genérico para a geração de dados de teste baseada em buscas

Ambos os conteúdos descritos nesta Seção foram apresentados em “SBST Aplicado ao Teste Estrutural” – (Bueno, 2011) em (Vergilio et al., 2011b). Espera-se que este conteúdo possa contribuir para a elaboração de novos trabalhos em SBST.

3.11. Considerações Finais

Este capítulo apresentou trabalhos que abordam a geração de dados de teste. Esta atividade visa selecionar dados de teste que exercitem os requisitos de teste, determinados pela utilização de um critério de teste.

Foi apresentada a Engenharia de Software Baseada em Busca – SBSE e foram descritas meta-heurísticas potencialmente aplicáveis a problemas de Engenharia de Software. Duas delas, o Recozimento Simulado e o Algoritmo Genético são utilizadas na técnica para a geração de dados proposta no próximo capítulo.

Foi dada ênfase à utilização de meta-heurísticas para a geração de dados de teste, área referida como Teste de Software Baseado em Buscas – SBST. Importante notar que o SBST é utilizado muitas vezes conjuntamente com a Abordagem Dinâmica de geração de dados, isto é, um modelo, ou o código é executado e as informações da execução são utilizadas pela meta-heurística para direcionar a busca por dados de teste que satisfaçam alguma propriedade desejada. Contudo, técnicas de processamento simbólico também podem ser utilizadas com esta finalidade.

Duas contribuições secundárias desta tese foram desenvolvidas e apresentadas na Seção 3.10:

- Pontos-chave organizados em um guia de sete passos genéricos a serem realizados para a concepção e definição de uma nova abordagem SBST.
- Definição de um procedimento genérico e simples para a geração de dados com o uso de meta-heurísticas.

Conforme destacado neste capítulo, pesquisadores da área SBST têm focado prioritariamente na automação da geração de dados de teste para exercitar os requisitos de técnicas e critérios de teste, como o teste baseado em estrutura, o teste baseado em modelos, ou o teste baseado em defeitos. Uma linha distinta de trabalhos SBST objetiva a avaliação de requisitos não funcionais do software em teste, como o tempo de execução, a qualidade de serviço, aspectos de segurança de informação, ou o comportamento em condições de exceção. Estes trabalhos podem ser considerados como exemplos de “aplicação tradicional” de SBST, isto é, utilizar meta-heurísticas para gerar dados para técnicas e critérios de teste já bem estabelecidos na literatura.

Uma área de investigação mais recente considera situações em que o teste é realizado com mais de um objetivo, muitas vezes objetivos conflitantes. Esses casos são tratados com

técnicas específicas para estas situações, como a otimização multiobjetivo (Zitzler et al, 2003).

As abordagens de teste multiobjetivo dão um passo além da “aplicação tradicional”. Nestes casos, os dados de teste gerados possuem atributos induzidos pelos vários objetivos da busca. Como há mais de um objetivo, os dados de teste apresentam características além das obtidas com a aplicação tradicional do SBST, que “somente” exercita elementos requeridos de técnicas e critérios. Por exemplo, dados de teste gerados na abordagem de Pinto e Vergilio (2010) podem conjuntamente: identificar problemas de funcionalidade (ao considerar o objetivo de maximizar os elementos do código exercitados) e minimizar o tempo de execução (ao considerar o objetivo de minimizar este parâmetro). De certo modo, estas abordagens buscam um balanceamento entre os objetivos, levando a conjuntos de teste com características únicas.

Apesar deste avanço com as abordagens de teste multiobjetivo, nota-se que as abordagens SBST estão ainda subutilizando o potencial oferecido pela meta-heurísticas como ferramentas de geração de dados de teste. Esta utilização oferece ao engenheiro de software a possibilidade de definir livremente propriedades-alvo de conjuntos de teste, que transcendem a meramente exercitar os requisitos de conhecidas técnicas e critérios de teste. Esta possibilidade existe devido a dois pontos:

1. Pode-se resolver problemas com meta-heurísticas pela descrição das características que se deseja para a solução do problema, sem a necessidade de indicar os passos necessários para que a solução seja alcançada (Von Zuben, 2000). Isto é, pode-se definir “o que” se deseja em um conjunto de teste, sem necessariamente se descrever “como” obtê-lo.
2. Essas ferramentas são muito robustas para a solução de problemas complexos. Isto permite um grande grau de liberdade sobre as informações a serem utilizadas para a geração de dados, bastando representá-las matematicamente na forma de uma função objetivo.

No próximo capítulo é apresentada uma técnica de geração de dados de teste que tem como alvo propriedades de conjuntos de teste. A ideia é aproveitar o potencial de meta-heurísticas para gerar dados de teste que apresentem propriedades que, espera-se, permitam um teste eficaz. Em relação à “aplicação tradicional” em SBST a diferença desta proposta é que as meta-heurísticas passam a ser uma ferramenta para explorar novas possibilidades em técnicas e critérios de teste, e não apenas para gerar automaticamente dados para exercitar elementos requeridos de técnicas e critérios conhecidos. Como propriedade alvo, foi escolhida a diversidade do conjunto de teste.

Capítulo 4 – A Diversidade Como uma Técnica de Teste de Software

“Highly repeatable testing can actually minimize the chance of discovering all the important problems, for the same reason that stepping in someone else’s footprints minimizes the chance of being blown up by a land mine.”

James Bach

Este capítulo apresenta a técnica Geração de Dados de Teste Orientada à Diversidade (*Diversity Oriented Test Data Generation – DOTG*) e as meta-heurísticas para a geração automática de dados para esta técnica.

Antes de apresentar a técnica proposta, é fornecida uma visão geral do conceito de diversidade em diferentes áreas de conhecimento, como em ciências sociais e em ciências biológicas. É abordado o conceito da diversidade no teste de software: é apresentada uma visão intuitiva do papel da diversidade no teste por meio de um exemplo. São descritas relações entre o conceito de diversidade e software, envolvendo, por exemplo, confiabilidade de software, tolerância a defeitos, e técnicas de teste baseadas em particionamento.

É apresentada então a proposta da Geração de Dados de Teste Orientada à Diversidade (DOTG). É descrito como diferentes variações da técnica DOTG podem ser criadas, dependendo do tipo de informação utilizada para se medir a diversidade. Cada variação é chamada de *Perspectiva* da diversidade, e é fornecido um guia com aspectos a serem considerados para se instanciar uma variação (perspectiva) da técnica DOTG. Este guia é referido como “meta-modelo de diversidade”.

A Perspectiva do Domínio de Entrada para a Geração de Dados de Teste Orientada à Diversidade (DOTG-ID) é descrita. São feitas definições de distância entre dados de teste e de diversidade de conjuntos de teste, e as características de conjuntos de teste de alta diversidade (DOTS) são descritas.

São detalhadas as meta-heurísticas implementadas para a geração automática de dados de teste de alta diversidade (DOTS): SA-DOTG, baseada em Recozimento Simulado; GA-DOTG, baseada em Algoritmo Genético; e a SR-DOTG, uma meta-heurística proposta nesta Tese, na qual partículas (dados de teste) eletricamente carregadas apresentam um comportamento emergente de auto-organização iterativa, espontaneamente gerando os conjuntos DOTS.

Por fim, são descritos outros trabalhos que guardam semelhanças com a Geração de Dados de Teste Orientada à Diversidade. São destacadas as semelhanças e os pontos nos quais a DOTG é única.

4.1. Uma Introdução Sobre o Conceito de Diversidade

A diversidade é definida como: “o estado ou fato de ser diverso”, ou “multiplicidade de diferenças” ⁵. Atualmente, o conceito de diversidade é visto em áreas como ciências sociais, biologia e engenharia. Embora a essência do significado de diversidade seja a mesma, os diferentes pontos de vista são particulares.

⁵ Fonte: dicionário Aurélio.

4.1.1. Diversidade nas Ciências Sociais – criatividade e inovação

Nas ciências sociais, diversidade refere-se à coexistência de várias entidades com diferenças em algum aspecto. Por exemplo, “diversidade cultural” faz menção à coexistência e respeito às “variadas formas de identidade de povos e culturas ao longo do tempo e do espaço, sua originalidade, singularidade, expressão”. São encontrados também termos como: “diversidade étnica” e “diversidade religiosa”, com significados próximos e que enfatizam a existência, em um determinado contexto, de indivíduos que manifestam variações em relação a etnia e religiosidade, respectivamente.

A diversidade, em seus vários aspectos, é muitas vezes vista como uma característica desejável em empresas e grupos de trabalho. A criatividade e inovação são favorecidas pela existência de uma multiplicidade de pontos de vista. Ou seja, as ideias de um grupo são potencialmente mais inovadoras e interessantes quando as pessoas trazem uma variedade de conhecimentos e perspectivas para a solução de um problema. Domenico De Masi (1995) destaca que “o confronto e a liberação recíproca das diferenças não deságuam automaticamente em conflito. Geralmente, o confronto das diferenças leva à integração das partes”.

Fischer (2005) relaciona diversidade e criatividade e destaca que, atualmente, a criatividade solitária é um mito. A colaboração é uma necessidade; portanto, compreender a criatividade individual assim como a criatividade social nos ambientes de trabalho é importante. A criatividade não acontece apenas dentro da cabeça das pessoas, mas também na interação entre os pensamentos das pessoas em um contexto sociocultural. A criatividade social de grupos colaborativos é impactada por aspectos de distância e diversidade de diferentes naturezas. O autor aborda diferentes tipos de distâncias que existem em projetos: distâncias físicas, temporais, tecnológicas e, sobretudo conceituais. Distâncias físicas, temporais e tecnológicas normalmente restringem a capacidade criativa do grupo e representam desafios. No entanto, a “diversidade de conceitos”, caracteriza grupos (comunidades) nas quais os indivíduos possuem diferentes experiências e interesses, e abordam os problemas a serem resolvidos sob diferentes perspectivas. Neste caso, a diversidade representa uma oportunidade para gerar novas ideias e novos ambientes.

Portanto, o desafio não é reduzir a heterogeneidade do grupo, mas apoiá-la, gerenciá-la, construir “pontes” entre diferentes fontes de conhecimento e explorar colisões e quebras conceituais como fontes para a inovação.

Portanto, sob o ponto de vista “social”, promover a diversidade entre indivíduos tem o potencial de gerar ambientes com maior potencial de inovação.

4.1.2. Diversidade na Química – representatividade

O conceito de diversidade na química, inicialmente, ganhou ênfase com o progresso de indústrias químicas e técnicas computacionais. Recentemente a área de química combinatória tem focado em desenvolver ferramentas e técnicas de software aplicadas a problemas de diversidade química em indústrias farmacêuticas e agroquímicas.

Uma tarefa fundamental associada à diversidade química é a de identificar um subconjunto restrito de componentes que melhor represente a totalidade de componentes existentes. As técnicas visam a geração de uma coleção de componentes químicos, a um custo limitado, mas com informação útil para ser utilizada como base para a exploração química de novos compostos. Esta diminuição de custo é conseguida justamente pela possibilidade de excluir componentes redundantes que não agreguem diversidade química ao conjunto, economizando assim com custos de processamento desses componentes. Exemplos de trabalhos que aplicam conceitos e ferramentas computacionais de diversidade para manutenção de bibliotecas químicas e outras aplicações: (Ostresh et al., 1994), (Willett, 2000), (Agrafiotis, 2002), (Barlett e Entzeroth, 2006).

4.1.3. Diversidade nas Ciências Biológicas – preservação e proteção

Na biologia, a diversidade pode ser vista como uma característica que favorece a preservação da vida. A diversidade genética de indivíduos de uma espécie, associada a mecanismos de seleção natural, contribui para a preservação e a evolução da espécie. Em outro nível, a diversidade de células protetoras em um sistema imune, associada a mecanismos de memória e generalização, aumenta a eficácia do combate a entidades

perigosas. Em ambos os casos, a existência da diversidade permite o surgimento de estratégias prudentes para a manutenção da vida.

A seguir, é feita uma breve descrição da influência da diversidade na evolução de espécies e na resposta imune de organismos vertebrados. Importante destacar que o objetivo da descrição é apenas apontar conceitos e mecanismos presentes nos modelos biológicos, que podem fornecer uma base para analogias com aspectos da técnica de teste proposta. Portanto, os conceitos relativos à biologia são descritos de maneira informal. No Apêndice C, analogias entre o teste e mecanismos do sistema imune são descritas.

Diversidade de Espécies

Nas ciências biológicas, o conceito de diversidade é muito associado à ecologia e à teoria da evolução das espécies (Darwin, 1872), (Darwin, 2003). O termo biodiversidade pode se referir a diferentes aspectos da variedade da vida. A riqueza de espécies consiste simplesmente no número de espécies que existem em uma região de interesse. O conceito de biodiversidade pode referir-se a níveis mais abrangentes e complexos da organização da vida. A diversidade pode referir-se, por exemplo, aos seguintes aspectos: a variedade genética existente dentro das espécies; a variedade de espécies da flora, da fauna, de fungos e de micro-organismos em um ambiente; a variedade de funções ecológicas desempenhadas pelos organismos nos ecossistemas; e a variedade de *habitats* e ecossistemas formados pelos organismos.

A diversidade existente entre os indivíduos de uma espécie tem impacto na evolução desta espécie. A evolução de espécies ocorre principalmente pela *seleção natural*, um processo pelo qual os indivíduos com características adequadas à vida em um ambiente possuem maior chance de sobreviver e de gerar descendentes do que os indivíduos menos aptos, com características menos adequadas ao ambiente. Deste modo, com o passar das gerações, as características hereditárias que contribuem para a sobrevivência – e reprodução – tornam-se mais comuns na espécie, enquanto que características prejudiciais tornam-se mais raras.

Neste processo evolutivo, a diversidade de material genético existente entre os vários indivíduos da espécie propicia um maior repertório de características físicas e comportamentais, e uma consequente maior capacidade de adaptação às mudanças que ocorram no ambiente. Isto é, uma maior quantidade de variação entre os indivíduos permite que a espécie se adapte mais rapidamente e de forma mais eficaz a novos *habitats*, diminuindo as chances de que a espécie se torne extinta devido às mudanças no *habitat* original.

Portanto, neste contexto, a diversidade pode ser vista como uma estratégia preventiva que aumenta a capacidade de uma espécie em tolerar as adversidades impostas pelo ambiente, levando a uma vantagem competitiva desta espécie em relação às outras espécies concorrentes existentes neste ambiente.

Diversidade no Sistema Imune

O *Sistema Imune* (ou *Sistema Imunológico*) é responsável por proteger seres vivos de entidades perigosas internas e externas, sendo fundamental para a manutenção da vida (De Castro, 2001), (De Castro e Timmis, 2002). Essas entidades perigosas, chamadas genericamente de Patógenos, são micro-organismos causadores de doenças, tais como vírus, bactérias, fungos, ou parasitas.

Existem vários mecanismos distintos, algumas vezes até redundantes, que se encarregam de defender os indivíduos contra entidades perigosas. Esses mecanismos podem agir contra uma entidade específica ou podem ser responsáveis por agir contra uma variedade de entidades.

Dentre as características e mecanismos do sistema imune, podem ser destacadas:

- *Barreiras físicas e bioquímicas*, tais como, respectivamente, a pele e os ácidos estomacais;
- Um *sistema imune inato*, cujas células mediadoras (macrófagos e granulócitos) apresentam uma resposta rápida e estão disponíveis para combater uma grande

variedade de patógenos. O sistema imune inato funciona de forma semelhante em todos os indivíduos normais.

- Um *sistema imune adaptativo*, capaz de fornecer uma resposta específica, produzindo *anticorpos* para um determinado patógeno. Células *linfócito B* e *linfócito T* produzem os anticorpos em resposta a um ataque específico (infecção) e desenvolvem uma memória imunológica. Portanto, caso o patógeno volte a entrar em contato com o organismo, o sistema imune reage prontamente, visto que já possui linfócitos ativos específicos para o patógeno. Esta resposta é constantemente aperfeiçoada a cada novo ataque do patógeno.

Os sistemas imunes (inato e adaptativo) possuem mecanismos de identificação capazes de distinguir células próprias saudáveis de patógenos (ou *antígenos*). Os mediadores da resposta imune são os linfócitos, que produzem anticorpos, capazes de reconhecer e eliminar os patógenos. Cada anticorpo possui, em sua superfície, *receptores* particulares para um tipo de patógeno, resultando em uma especificidade para a ligação (ou reconhecimento) anticorpo-antígeno. O *reconhecimento imune* ocorre quando o nível de complementaridade entre o receptor do anticorpo e o *epitopo* do antígeno (parte do antígeno que se liga ao anticorpo) está acima de certo limiar. Este reconhecimento é fundamental para o sucesso do combate ao patógeno. Uma analogia para esta especificidade antígeno-anticorpo no reconhecimento imune é a especificidade existente entre uma fechadura e a sua chave.

Os recursos para esta atuação do sistema imune são limitados: os tipos de anticorpos do repertório são finitos. No entanto, este sistema precisa reconhecer um número potencialmente infinito de patógenos e células cancerígenas. O desenvolvimento de estratégias para uma utilização “inteligente” desses recursos limitados, que permita um funcionamento eficaz do sistema, é uma consequência da evolução das espécies por seleção natural.

Diversidade nas Ciências Biológicas: Sumário

Esta seção apresentou mecanismos de preservação da vida relacionados à diversidade. Foi apresentada a diversidade entre os indivíduos de uma espécie como fator importante para a adaptação às mudanças no meio e a manutenção da espécie. Foram descritas características do sistema imune e foi destacado o papel da diversidade de anticorpos no sucesso ao combate de agentes perigosos.

Esta seção não detalhou trabalhos que analisam a estratégia utilizada pelo sistema imune para cobrir o espaço de possíveis patógenos. Este problema de cobertura e a estratégia utilizada por este sistema guardam semelhanças com o problema de cobertura do domínio de entrada de um programa em teste. Esta analogia é explorada no Apêndice C.

4.2. A Diversidade no Teste de Software

Esta seção aborda a utilização do conceito de diversidade para as atividades de teste de software. Inicialmente, é apresentada uma visão informal e intuitiva da relação entre diversidade e teste. Em seguida, são resumidos conceitos que relacionam diversidade e software.

4.2.1. Uma Visão Intuitiva da Diversidade

Quando uma tarefa de projetar casos de teste é alocada a um(a) analista de teste, é esperado que ele(a) crie um conjunto de casos de teste (e procedimentos de execução desses casos de teste) que sejam abrangentes e representativos do conjunto de funcionalidades do software. Espera-se que a execução deste conjunto de casos de teste permita revelar defeitos associados às diferentes funcionalidades e também ganhar confiança de que o software satisfaz os seus requisitos.

Naturalmente, pode haver a necessidade de um teste mais restrito, abordando apenas um subconjunto das funcionalidades do software ou alguma característica específica. Caso

haja algum motivo para focar a atenção em certa funcionalidade do software, ou em certa parte específica do código, este objetivo deve ser explicitado nas fases de planejamento do teste e de projeto do teste. Isto ocorre, por exemplo, quando o objetivo é avaliar unicamente uma parte do código que sofreu mudanças recentes, ou quando se deseja avaliar se certas características desejáveis estão presentes ou não no software (requisitos não funcionais, tais como: nível de desempenho, segurança, capacidade de carga, facilidade de uso, etc.).

No entanto, na maior parte dos casos, há a necessidade de se testar o software de uma maneira abrangente. Informações sobre o software ⁶ devem ser consideradas para a criação de um conjunto de casos de teste que exercitem o software tão completamente quanto possível, considerando as limitações dos recursos disponíveis. Neste contexto, recursos dizem respeito ao tempo e ao esforço necessários para definir os dados de teste e os respectivos resultados esperados (os casos de teste); executar esses casos de teste no software sob teste; e avaliar as saídas produzidas.

Uma premissa fundamental para a estratégia de criar casos de teste com alta variedade – ou diversidade – é que não se saiba a priori com grande confiança qual parte do software (em termos de funcionalidades, ou de módulos de código fonte) possui defeitos associados. Caso esta informação sobre a localização de defeitos esteja disponível, devem ser realizados testes específicos, restritos às funcionalidades associadas aos defeitos (ou restrito a módulos do código fonte propensos a tê-los) visando revelá-los e removê-los do código. Neste caso, pode-se também realizar a inspeção do código fonte para a identificação e a correção do defeito. É importante destacar que, na prática, raramente se sabe com segurança onde estão os defeitos do software; portanto, a premissa anterior é bastante realista e pouco restritiva.

Como uma ilustração, vamos considerar a situação levantada no início desta seção: alocar aos analistas de teste a tarefa de definir conjuntos de casos de teste para um software. Vamos supor ainda que se trata de um *software de apoio ao funcionamento de bibliotecas*, com funcionalidades para o cadastro do acervo; o cadastro de funcionários da biblioteca; o

⁶ Informações podem ser relacionadas aos requisitos do software, a modelos construídos ao longo do desenvolvimento ou ao código fonte.

cadastro de usuários da biblioteca; e o registro de empréstimos, devoluções e reservas. Cada uma dessas funcionalidades naturalmente possui uma série de detalhamentos sobre informações a serem fornecidas, variações de comandos a serem acionados, e o comportamento esperado do sistema em cada situação. Um(a) analista de teste “A”, ao se confrontar com esta tarefa, gastou o tempo disponível para criar um conjunto de 100 casos de teste que avaliem minuciosamente a funcionalidade de empréstimos de livros, incluindo casos de teste para: realizar diversos empréstimos de livros diferentes; empréstimos de diferentes exemplares de um mesmo livro; empréstimos para vários diferentes usuários com pendências, etc. Um(a) analista de teste “B” gastou o tempo disponível para criar também 100 casos de teste; mas “B” distribuiu os casos de teste nos vários módulos, exercitando os aspectos principais de cada um dos módulos. Os casos de teste de “B” envolvem a inclusão e consulta de um funcionário, inclusão e consulta de livros e periódicos, cadastro de usuários, e operações de reserva, empréstimo e devolução de itens.

A situação descrita no parágrafo anterior busca ilustrar, de maneira informal, uma intuição relacionada às atividades de teste: considerando que o objetivo do teste seja avaliar o software de uma maneira global, a variabilidade/diversidade dos casos de teste utilizados é um aspecto importante a ser considerado para a seleção ou avaliação de um conjunto de dados de teste. Ao se adotar uma estratégia “conservadora” e precavida, não é adequado selecionar vários dados de teste muito semelhantes entre si; estes dados exercitam bem uma pequena parte do software, deixando as partes restantes pobremente exercitadas e mal testadas. Este foi o comportamento do (a) analista “A”, que criou um conjunto com vários casos de teste semelhantes e exercitou cuidadosamente a funcionalidade de empréstimos de livros. Se “A” fez um “bom trabalho” no projeto desses casos de teste, provavelmente esta funcionalidade foi bem exercitada, embora isto não seja garantido. Por exemplo, “A” poderia, com o objetivo de terminar rápido o trabalho, criar 100 casos de teste praticamente idênticos, todos realizando um empréstimo de algum livro disponível para algum usuário sem pendências. Neste último caso, mesmo esta única funcionalidade considerada no teste teria sido mal testada, devido à pouca variedade dos dados de teste utilizados. Obviamente, como as outras funcionalidades do software não foram testadas (além da funcionalidade

“empréstimos de livros”), os defeitos nelas existentes, mesmo que “grosseiros”⁷, não têm como ser descobertos.

Ainda sobre esta situação ilustrada, o analista “B” utilizou os recursos de teste para exercitar cada um dos módulos, e teve chance de encontrar defeitos em todos os módulos. No entanto, como os casos de teste foram distribuídos entre estes módulos, é possível que nenhum deles tenha sido exercitado de forma severa o suficiente e, portanto, alguns defeitos podem não ter sido descobertos.

O que se buscou ilustrar no exemplo anterior é que a qualidade de um teste realizado pode ser avaliada (ou presumida) por meio da análise conjunta de dois fatores:

- O número de casos de teste utilizados;
- A diversidade – ou variedade – dos casos de teste utilizados.

O primeiro fator (número de casos de teste) pode ser entendido por meio do seguinte raciocínio: se o software possui defeitos, existe uma chance maior do que zero de que o software falhe ao ser executado com um caso de teste. Portanto, cada novo caso de teste contribui para aumentar a chance de que o software em teste falhe; quanto maior o número de casos de teste, maior será a chance de ocorrerem falhas.

O segundo fator (diversidade dos casos de teste), por sua vez, pode ser compreendido a partir da seguinte ideia: se não há certezas sobre a ausência de defeitos em alguma parte do software, deve-se distribuir os casos de teste de forma a exercitar todas as funcionalidades (ou módulos) que estão no escopo do teste. Neste ponto, identifica-se uma relação entre estes dois fatores: para que um teste abrangente (de alta diversidade) exercite satisfatoriamente o software, é necessário que o número de casos de teste seja adequado (ou grande o suficiente) para assegurar um bom teste⁸.

⁷ O termo “grosseiros” visa caracterizar esses defeitos como sendo relativamente fáceis de serem descobertos; por exemplo, por estarem associados a funções do software muito utilizadas, ou ainda por estarem associados a uma grande porção do domínio de entrada – o conceito de defeitos semanticamente grandes.

⁸ Neste contexto, não vamos nos estender na discussão de como encontrar um tamanho adequado para o conjunto de teste. Embora este seja um ponto importante associado a técnicas de teste – quando se deve parar de testar –, não é o foco deste trabalho.

Importante notar também que esses dois fatores, relativos aos dados de teste, devem ser considerados levando-se em conta o objeto do teste. A complexidade do software em teste, avaliada do ponto de vista dos requisitos ou do ponto de vista do código fonte, em última análise, determina o número de casos de teste e o nível de diversidade recomendados.

Profissionais de teste de software geralmente usam as suas experiências anteriores e a intuição para identificar um grande número de variações de situações a serem avaliadas no teste. Esta é uma diretriz utilizada informalmente e que encontra respaldo em registros de “melhores práticas”. Conceitos relacionados à diversidade/variação estão subjacentes em algumas técnicas de teste e também em áreas como confiabilidade de software e tolerância a defeitos.

Exemplos de ideias relacionadas à importância da variação/diversidade no teste:

- Uma das 28 melhores práticas que contribuem para a qualidade do teste de software, em um relatório da IBM, aborda a criação dos casos de teste: “Criar diferentes variações de testes para cobrir o espaço de estados do programa tanto quanto seja necessário” e “Compreender como criar variações de dados de teste e alcançar uma cobertura que seja adequada o suficiente para testar de forma completa uma função” (Chillarege, 1999).
- O Teste Exploratório é caracterizado pelo “simultâneo aprendizado, projeto e execução”. Isto é, o testador realiza o projeto dos testes e, à medida que esses testes são executados, a informação obtida é usada para projetar novos testes e testes melhores (Bach, 2003), (Bach, 2005). Segundo Kaner et al (2001), o teste exploratório se caracteriza por: interatividade, concorrência de cognição e execução; e orientação a resultados rápidos. Uma ideia essencial é a confiança na “habilidade dos testadores de usar heurísticas e experiências prévias para criar ideias diversas para o teste” (Kaner et al., 2001).
- O “paradoxo do pesticida” descrito por Beizer (1990): “todo método usado para prevenir ou encontrar defeitos deixa um resíduo de defeitos sutis para os quais esses

métodos são ineficazes”. Essa afirmação faz menção à possibilidade de uma fração da população de insetos desenvolver tolerância ao pesticida específico, fazendo com que ele seja ineficaz se usado repetidamente nesta população.

A analogia é que, ao longo do teste, o software pode tornar-se “imune” aos testes realizados, de forma análoga à imunidade adquirida pelos insetos ao pesticida. O paradoxo do pesticida é frequentemente associado à ideia de que as técnicas de teste tendem a revelar defeitos de determinados tipos e que, portanto, a utilização de diferentes técnicas no teste de um software tende a aumentar o número de defeitos revelados. Utilizar diferentes técnicas de teste para revelar mais defeitos seria equivalente a utilizar diferentes pesticidas para matar mais insetos.

Uma maior diversidade existente nos insetos da população tende a propiciar uma maior chance de que alguns insetos resistam ao pesticida. Do ponto de vista do fazendeiro, a informação útil é que variar o pesticida usado tende a melhorar a eficácia no extermínio dos insetos, visto que o novo pesticida pode atuar nos insetos que toleraram o pesticida anterior. Analogamente, do ponto de vista do teste, a informação útil é que assegurar uma alta variação/diversidade dos novos cenários de teste, em relação aos cenários testados anteriormente, tende a aumentar a chance de revelar novos defeitos no software, “imunes” aos testes anteriores.

- Na mesma linha da ideia anterior de Beizer, mas de maneira formal, Littlewood et al (2000) modelam o efeito de combinar técnicas de teste diversas. Adotar uma segunda (ou terceira) técnica para selecionar casos de teste é uma maneira bem conhecida de introduzir esta variação/diversidade no teste. Littlewood et al mostram que, quando se utilizam conjuntamente várias técnicas diferentes, a eficácia do teste depende da interação de dois aspectos: a eficácia individual de cada técnica; e a dependência entre as técnicas. Por meio de medidas de eficácia para revelar defeitos e de diversidade de técnicas, os autores identificaram que o efeito de aplicar repetidamente técnicas similares não aumenta significativamente a eficácia do teste. Ao contrário, combinar técnicas de teste diversas tende a ampliar

significativamente esta eficácia. A conclusão é de que aplicar conjuntamente técnicas de teste diversas é uma boa prática, a ser utilizada sempre que possível.

4.2.2. Diversidade, Tolerância a Defeitos, Confiabilidade, Particionamento e Cobertura

Outros conceitos que de algum modo relacionam diversidade e software, sob a ótica do teste de software:

- **Diversidade e tolerância a defeitos.** A Tolerância a Defeitos (*fault tolerance*)⁹ visa a garantir a continuidade de um determinado serviço, mesmo na presença de defeitos que causem estados de erro na execução do software (Randel, 1975), (Murugesan, 1989). A abordagem “software com N-versões” para a tolerância a defeitos tem a sua eficácia associada à promoção da diversidade entre as várias versões a fim de evitar falhas coincidentes das versões (Avizienis e Kelly, 1984), (Avizienis, 1985), (Littlewood et al., 2001). Outro trabalho na mesma linha, de Ammann e Knight (1988), usa o conceito de Diversidade de Dados para que certas classes de defeitos de software sejam toleradas por meio de uma re-expressão e re-execução das entradas do software. Neste caso, busca-se com a re-expressão retirar o dado de teste da região do domínio que causa falhas. Essas abordagens utilizam a diversidade para “evitar falhas”, enquanto que a técnica de teste proposta a utiliza com o objetivo oposto. Um ponto que une as duas áreas (diversidade para tolerar defeitos e diversidade no teste) é que o processo que acarreta a falha em software é o mesmo (Thompson et al., 1993), (Voas, 1992), (Ehrlich et al., 1991).
- **Diversidade e confiabilidade de software.** A confiabilidade de um software é definida como “a probabilidade de um sistema de software operar sem falhas por um período de tempo específico e sob condições de operação especificadas.” (Littlewood e Strigini, 2000), (Lyu, 2007), (IEEE, 1990). Whittaker e Voas (2000)

⁹ Trata-se de uma área comumente designada como “tolerância a falhas”. Utiliza-se a tradução do inglês (tolerância a defeitos) para manter a uniformidade das definições utilizadas.

em uma avaliação de novas direções para pesquisa em modelos de confiabilidade afirmam: “deve-se garantir que os dados de teste sejam diversos o suficiente para formar um conjunto completo de dados de teste”. Esta conclusão é resultado da análise de que considerar o tempo de operação do software em teste não faz sentido em confiabilidade de software (trata-se de uma herança de modelos de confiabilidade de hardware). Isto é, mais importante do que o tempo em que software foi utilizado, é como o teste exercitou o software.

Outros trabalhos destacam que a cobertura estrutural alcançada pelos dados de teste tem maior correlação com a confiabilidade do software do que o tempo de operação e o número de casos de teste executados (Chen et al., 2001), (Malaiya et al., 2002), (Crespo et al., 2008). No Apêndice C é desenvolvida a ideia de que casos de teste que permitem melhores estimativas de confiabilidade do software caracterizam-se por serem complementares aos já existentes no conjunto de teste (eles exercitam requisitos/funcionalidades não ainda exercitadas e permitem obter “nova informação” sobre o software). Esta busca de dados de teste minimamente sobrepostos e mutuamente complementares é uma característica da técnica de teste proposta nesta tese.

- **Diversidade e teste baseado em particionamento.** O teste baseado em particionamento abrange um conjunto de técnicas que divide o domínio de entrada do programa em partes (partições, ou subdomínios) e requer que pelo menos um dado de teste de cada partição seja executado (Hamlet e Taylor, 1990), (Ntafos, 1988) (Chen e Yu, 1994). Weyuker e Jeng (1991) abordam analiticamente a capacidade de revelar defeitos dessas técnicas, comparadas ao teste aleatório. São investigados fatores que tornam essas técnicas mais eficazes (ou menos) do que o teste aleatório (fatores relacionados à taxa de falhas e à forma de divisão do domínio). Gutjahr (1999) estende o modelo analítico de Weyuker e Jeng considerando os *valores esperados* das taxas de falhas para subdomínios ao invés de considerar *valores definidos* para taxas de falhas. O argumento para esta extensão é que as taxas de falhas associadas aos subdomínios nunca são conhecidas com

certeza antes do teste. Os resultados mostram que ao se considerar a *incerteza* sobre a taxa de falhas de cada subdomínio, o teste baseado em particionamento tende a ser amplamente mais eficaz do que o teste aleatório na maior parte dos casos. Segundo análise de Gutjahar as técnicas baseadas em particionamento induzem a uma diversificação dos dados de teste. Esta diversificação torna-se vantajosa para a eficácia do teste quando as taxas de falha dos subdomínios – ou analogamente a localização dos defeitos – são desconhecidos. O autor faz um paralelo com decisões de investimentos em situações de incertezas do retorno, ocasião em que a diversificação tende a ser a estratégia com melhor retorno. Esta ênfase de que diversificar o teste permite aumentar a capacidade de revelar defeitos é um ponto importante da técnica de teste proposta.

- **Diversidade como estratégia de cobertura do sistema imune.** Conforme já descrito neste capítulo, cabem analogias entre estratégias do sistema imune e técnicas de teste. Ambos os problemas (o reconhecimento imune por meio de anticorpos, e a revelação de defeitos por meio de dados de teste) podem ser vistos como problemas de cobertura em espaços de soluções. Embora seja uma área largamente distinta do teste de software, com mecanismos biológicos e químicos particulares, estas analogias permitem enxergar a técnica de teste proposta por um ponto de vista diferente. Isto pode ser útil especialmente para inspirar extensões e aprimoramentos da técnica.

O Apêndice C descreve detalhadamente como cada aspecto listado acima se relaciona com o teste de software de modo geral, e mais especificamente, como esses conceitos embasaram a definição da Geração de Dados de Teste Orientada à Diversidade, descrita a seguir.

4.3. Geração de Dados de Teste Orientada à Diversidade

Nas seções anteriores, buscou-se mostrar as ideias que motivaram e direcionaram a definição do Teste Orientado à Diversidade. A seguir, discute-se como caracterizar

diversidade de dados de teste e são apresentados os conceitos de perspectivas da diversidade e um meta-modelo para critérios de teste orientados à diversidade.

4.3.1. A Diversidade Ideal sob a Ótica de Técnicas de Teste

Ao se remeter à definição de que diversidade é a “multiplicidade de diferenças” e levando em conta o foco em técnicas de teste de software, o questionamento evidente é: como caracterizar um dado de teste, de forma a diferenciá-lo dos demais dados de teste de uma maneira coerente? Identificar diferenças que sejam relevantes entre os dados de teste é um ponto importante no desenvolvimento da técnica de teste proposta. Considerando o objetivo de revelar defeitos no teste, esta questão pode ser vista como semelhante à questão levantada por Hamlet (2002) na discussão sobre continuidade em sistemas de software: “se o software funcionou corretamente com uma entrada x_1 , ele deve também funcionar corretamente para uma entrada na vizinhança de x_1 . Se fosse possível determinar a dimensão desta vizinhança, uma segunda entrada x_2 a ser testada deveria estar obviamente fora da vizinhança de x_1 . Neste caso, se o software funcionou corretamente para x_1 e se é possível garantir que na vizinhança Δ ele se comportará do mesmo modo, não seria necessário testá-lo para nenhum valor $x_1 \pm \Delta$, pois se garantiria que ele também funciona corretamente para qualquer valor nesta região. Neste caso, só faz sentido um teste x_2 se $x_2 > x_1 + \Delta$ ou $x_2 < x_1 - \Delta$.

Um conjunto hipotético e ideal de dados de teste de alta diversidade tem como características:

- a) Cada dado de teste x_i possui uma vizinhança Δ_i para a qual se garante que o software funcione corretamente (considerando que o funcionamento correto do software com x_i foi evidenciado);
- b) Cada dado de teste x_i situa-se fora da vizinhança de qualquer outro dado de teste do conjunto;

- c) A união de todas as vizinhanças Δ_i de todos os dados de teste x_i inclui todas as possíveis situações para a execução do software. Ou seja, não existe uma situação possível que não pertença a pelo menos uma vizinhança Δ_i .

Seguindo a lógica acima, ter-se-ia uma forma ideal de caracterizar diferenças relevantes entre dados de teste e a diversidade de conjuntos de dados de teste. Porém, este raciocínio tem um caráter apenas ilustrativo e conceitual, pois do ponto de vista prático é impossível determinar essas vizinhanças e assegurar que elas incluam todas as possíveis situações de execução – assim como é impossível garantir que as regiões de falha sejam agrupadas em partições nas técnicas baseadas em particionamento.

Portanto, como uma abordagem prática, decidiu-se explorar alternativas para caracterizar a diversidade do teste em diferentes perspectivas.

4.3.2. Perspectivas e um Meta-Modelo Para a Diversidade

Diferentes variações do teste orientado à diversidade podem ser definidas por meio da consideração de diferentes perspectivas (ou aspectos) do software. Exemplos de aspectos:

- Os requisitos funcionais do software;
- Os requisitos não funcionais do software;
- O comportamento esperado do software (ex: especificado na forma de modelos);
- As entradas recebidas pelo software – domínio de entrada;
- Os resultados produzidos pelo software – domínio de saída; e
- A implementação do software (ex: informações sobre a execução do software, tais como fluxos de controle e fluxos de dados).

Esses critérios, independentemente da sua perspectiva, são referidos genericamente como “Geração de Dados de Teste Orientada à Diversidade” (*Diversity Oriented Test Data Generation – DOTG*). Os conjuntos de dados de teste gerados por esses critérios são

referidos como “Conjuntos de Teste Orientados à Diversidade” (*Diversity Oriented Test Sets – DOTS*).

Neste trabalho, é desenvolvida a perspectiva do domínio de entrada do software para a diversidade (*DOTG-ID*, *ID* refere-se a *Input Domain*): considera a diversidade de alocação de dados de teste no domínio de entrada do software, refletida nos valores das variáveis de cada dado de teste. Os conjuntos de dados de teste são chamados Conjuntos de Teste Orientados à Diversidade, perspectiva do domínio de entrada (*DOTS-ID*), ou simplesmente conjuntos de dados de teste de alta diversidade (*DOTS*). A partir deste ponto, assume-se conjuntos *DOTS* como sendo *DOTS-ID*, a não ser que explicitado o contrário.

Para todas as perspectivas da diversidade, pode-se caracterizar aspectos comuns requeridos para viabilizar a obtenção de conjuntos de dados de teste. Esses aspectos representam um “meta-modelo” para diversidade, visto que eles definem aspectos genéricos a serem considerados para o desenvolvimento de uma técnica de teste orientado à diversidade. Esses aspectos são descritos a seguir.

- a) Definir uma métrica (medida) de diversidade que quantifique a diversidade de cada conjunto de dados de teste utilizando informações sobre os dados de teste, e/ou informações sobre a execução do software com os dados de teste, e/ou informações sobre a execução de modelos com os dados de teste.
- b) Definir um procedimento pelo qual a diversidade de certo conjunto de dados de teste possa ser calculada.
- c) Definir um algoritmo que permita gerar automaticamente dados de teste que apresentem altos níveis de diversidade – considerando medidas definidas em (a). Esses algoritmos podem utilizar conjuntos de dados de teste gerados aleatoriamente, como ponto de partida (referidos como Conjuntos Aleatórios de Teste ou *Random Test Sets – RTS*). Esses conjuntos são transformados, de forma iterativa, com o objetivo de aumentar as suas diversidades. O procedimento em (b) é utilizado para avaliar a diversidade de cada conjunto de dados de teste e, eventualmente, para direcionar as transformações a serem realizadas para aumentar a diversidade. Esses

algoritmos geram, como resultado, conjuntos de dados de teste de alta diversidade (*DOTS*).

Os aspectos **a**, **b**, **c** podem ser vistos como um guia para se desenvolver novas perspectivas para o teste orientado à diversidade. Um ponto ausente nesses aspectos diz respeito à avaliação do critério. Este ponto é descrito no aspecto (**d**) a seguir.

d) Definir um procedimento de avaliação da técnica de geração de dados de teste. Este procedimento deve estabelecer atividades de avaliação, assim como métricas a serem usadas, ambas específicas para cada característica a ser avaliada. Esta avaliação pode ser realizada pela comparação de características dos conjuntos de dados de teste aleatórios, utilizados como ponto de partida (veja aspecto **c**) com as características dos conjuntos de dados de teste de alta diversidade. Comparações podem ser realizadas por meio de simulações, ou por meio da execução do teste de programas selecionados, utilizando conjuntos de dados de teste gerados.

No caso do teste de programas, podem ser utilizadas métricas para estimar a eficácia da técnica de teste. Por exemplo, os valores de cobertura atingida segundo critérios estruturais (ex: percentual de ramos exercitados no critério Todos os Ramos, percentual de associações exercitados no critério Todos os Usos) ou segundo critérios baseados em defeitos (*score* de mutação no critério Análise de Mutantes). O procedimento de avaliação neste caso é composto por atividades para: executar o software com conjuntos de teste aleatório (RTSs); executar o software com conjuntos de teste orientados à diversidade (DOTSs); avaliar a cobertura atingida pelos RTSs; avaliar cobertura atingida pelos DOTSs; e analisar estatisticamente os dados de cobertura.

Os aspectos **a**, **b**, **c** e **d** podem ser vistos como um modelo genérico para se desenvolver e avaliar técnicas de geração de dados orientadas à diversidade.

Outro aspecto genérico, que independe da perspectiva da diversidade, diz respeito ao modo de uso da técnica de teste. Em geral, técnicas e critérios de teste podem ser aplicados com dois propósitos distintos, ainda que relacionados: avaliar dados de teste, ou selecionar

dados de teste (Rapps e Weyuker, 1985), (Maldonado, 1991), (Frankl e Weyuker, 1993). Variações da técnica DOTG, independentemente da perspectiva que adotam, devem idealmente poder ser aplicadas para os dois propósitos descritos anteriormente.

4.4. Perspectiva do Domínio de Entrada Para a Geração de Dados de Teste Orientada à Diversidade

Esta seção apresenta a perspectiva do domínio de entrada do programa em teste para a diversidade. São feitas definições iniciais; são descritas as funções de distância e de diversidade; e é descrito o mapeamento feito entre o domínio de busca das meta-heurísticas utilizadas para a geração de dados de teste e o domínio de entrada do programa.

4.4.1. Definições Iniciais

A conjectura inicial para o desenvolvimento da perspectiva do domínio de entrada da Geração de Dados de Teste Orientada à Diversidade (*DOTG-ID*) é que selecionar amostras do domínio de entrada do software de uma forma sistemática, usando uma alta diversidade de valores para as variáveis de entrada, tende a aprimorar a qualidade do teste em comparação à seleção puramente aleatória de amostras.

Refere-se a uma unidade de software (função ou procedimento) como um programa P . Uma variável de entrada para P pode ser um parâmetro de entrada, uma variável global usada em P , ou uma variável presente em um comando de entrada (*read(x)*). O domínio de entrada Dx_i da variável de entrada x_i é o conjunto de todos os valores que x_i pode ter. O domínio de entrada de P é o produto cartesiano dos domínios das variáveis de entrada. $D = Dx_1 \times Dx_2 \dots \times Dx_n$, onde n é o número de variáveis de entrada. Um dado de teste, ou dado de entrada t , é a combinação de valores associados às variáveis de entrada ($t = x_1, x_2, \dots, x_n$). Cada x_i representa uma variável de entrada, e t representa um único ponto no domínio de entrada n -dimensional D . Um conjunto de dados de teste (ou conjunto de teste) é uma

coleção de dados de teste ($T = t_1, t_2, \dots, t_m$); cada t_i representa um dado de teste e m é o tamanho do conjunto de dados de teste (ou $|T|$).

As definições anteriores são baseadas em trabalhos anteriores em geração de dados de teste (Korel, 1990). Algumas considerações práticas adicionais são necessárias:

- Assume-se que as variáveis de entrada são dos tipos primitivos presentes em linguagens de programação: inteiro, real, lógico e caractere. Vetores e Matrizes podem ser tratados pelos elementos que os constituem, por exemplo, um programa que recebe como entrada uma matriz bidimensional 3×3 de inteiros, pode ter sua interface vista como sendo uma sequência de 9 inteiros a serem associados à matriz em uma ordem predefinida. Registros podem ser tratados de forma análoga, isto é, cada campo do registro pode ser visto como uma variável independente. O tratamento de interfaces com elementos não primitivos requer *drivers* que mapeiem as variáveis primitivas para os tipos complexos recebidos como entrada pelo programa em teste.
- É possível que, em certos programas, a execução seja afetada por algum estado do ambiente (por exemplo, configurações de navegadores), ou pelo estado de variáveis perenes (por exemplo, pelo conteúdo de uma base de dados). Nestas situações, pode ser necessário que os estados do ambiente e da base de dados sejam tratados como variáveis de entrada adicionais, que devem compor o dado de teste.
- Em determinados tipos de software, pode ser necessário estender o conceito de dado de teste. Por exemplo, em software de tempo real, um dado de teste pode ser uma sequência de valores de entrada, cada valor associado a um ponto específico no tempo. Em um software interativo usando interface gráfica, um dado de teste pode ser uma sequência de valores e eventos, cujo tamanho pode variar de um dado de teste para outro.

As considerações anteriores são genéricas, relativas ao problema da geração de dados de teste. Considerações específicas sobre a técnica *DOTG* são descritas na Seção 4.4.3.

4.4.2. Distância e Diversidade

Considere dois dados de teste ta e tb : $ta = ta_1, ta_2, \dots, ta_n$; $tb = tb_1, tb_2, \dots, tb_n$. A distância entre dois dados de teste é definida por:

$$D(ta, tb) = \sqrt{\sum_{i=1}^n (ta_i - tb_i)^2} \quad (4.1)$$

$D(ta, tb)$ representa a *Distância Euclidiana* entre os dados de teste ta e tb no domínio de entrada n -dimensional do programa.

Considere um conjunto de dados de teste T formado de m dados de teste t : $T = t_1, t_2, \dots, t_m$. A diversidade de T é definida por:

$$Div(T) = \sum_{i=1}^m \min_j (D(t_i, t_j)) \quad (4.2)$$

Onde $\min_j (D(t_i, t_j))$ retorna o valor da distância entre t_i e t_j , sendo t_j o dado de teste mais próximo de t_i .

Portanto, o valor da diversidade de um conjunto de dados de teste T ($Div(T)$) é a soma das distâncias euclidianas entre cada dado de teste do conjunto e o seu vizinho mais próximo.

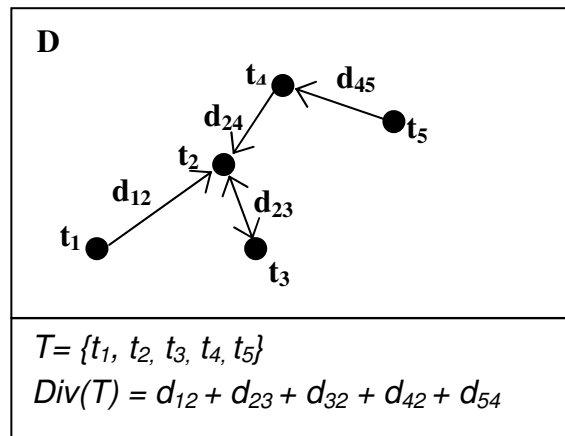


Figura 20. Exemplo de cálculo da diversidade

A Figura 20 mostra um domínio de entrada D com duas dimensões de um programa hipotético; cada dado de teste é definido por um par de valores (exemplo: x,y). O conjunto de dados de teste T é formado pelos dados de teste: $t_1, t_2, t_3, t_4, e t_5$; cada t_i representa um ponto no domínio 2-dimensional D . O valor da diversidade de T ($Div(T)$) é a soma das distâncias $d_{12}, d_{23}, d_{32}, d_{42}$ e d_{54} que separam cada dado de teste do seu vizinho mais próximo.

Importante notar que cada novo dado de teste adicionado a um conjunto de teste acarreta um aumento da diversidade do conjunto. Este aumento ocorre com um valor igual à distância do novo dado de teste ao dado de teste mais próximo já existente no conjunto de teste. Só não há aumento de diversidade se o novo dado de teste coincidir com algum dado de teste já executado. Isto faz sentido, pois, repetir a execução de um mesmo dado de teste não agrega “nova informação” sobre o software: se ele não falhou na primeira execução, não falhará na segunda, considerando que não sejam alterados o software nem o ambiente de teste.

A Figura 21 mostra três diferentes alocações de dados de teste em um mesmo domínio bidimensional ($a \times a$). Na Situação 1, dois dados de teste estão sobrepostos no canto inferior direito do domínio de entrada; na Situação 2, esses dois dados de teste estão próximos um do outro (a uma distância de $a/8$); na Situação 3 cada dado de teste situa-se distante dos demais e “explora” uma parte diferente do domínio de entrada.

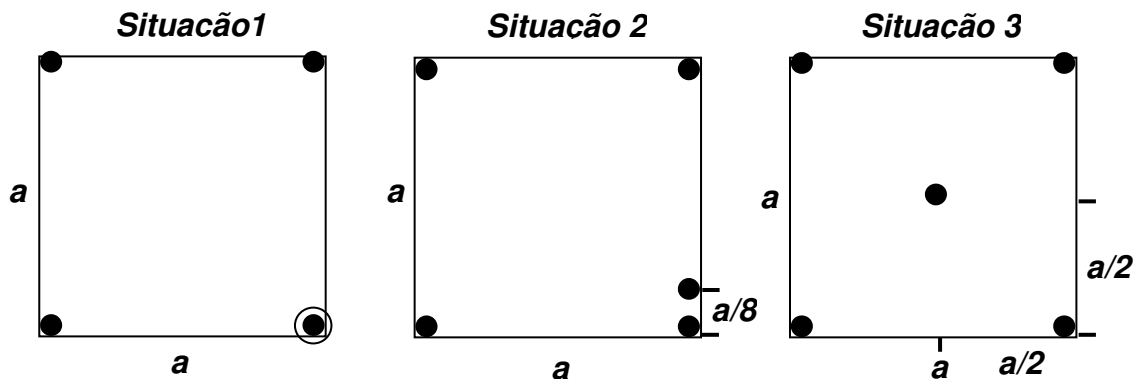


Figura 21. Alocação de dados de teste e valores de diversidade do conjunto de teste

Os valores de diversidade computados – utilizando Equação 4.2 – para estas situações são os seguintes (valores numéricos para $a = 1$):

- Situação 1 – $3a$ (3.000);
- Situação 2 – $25a/8$ (3.125);
- Situação 3 – $5a\sqrt{2}/2$ (3.536)

Ainda considerando-se as situações presentes na Figura 21, a Tabela 1 mostra os valores de diversidade ($Div(T)$, Equação 4.2) e também os valores resultantes da soma das distâncias de cada dado de teste para todos os outros dados de teste do conjunto – valor $SomaDist(T)$, Equação 4.3 abaixo. Notar que estes valores diferem dos valores de diversidade por somar (para cada dado de teste) todas as distâncias, ao invés de somar apenas a distância para o dado de teste vizinho mais próximo.

$$SomaDist(T) = \sum_{i=1}^m \sum_{j=1(i \neq j)}^m D(t_i, t_j) \quad (4.3)$$

Tabela 1. Valores de diversidade e de somatório de distâncias

	Div(T)	<u>SomaDist(T)</u> (obs: valores para $a=1$)
<u>Situação 1</u>	$3a$ (3.000)	$12a + 6a\sqrt{2}$ (20.485)
<u>Situação 2</u>	$\frac{25a}{8}$ (3.125)	$\frac{33a}{4} + 4a\sqrt{2} + \frac{a\sqrt{113}}{4} + \frac{a\sqrt{65}}{4}$ (20.329)
<u>Situação 3</u>	$\frac{5a\sqrt{2}}{2}$ (3.536)	$8a + 8a\sqrt{2}$ (19.314)

Ao se avaliar os valores $SomaDist$, fica clara a influência de se considerar apenas as distâncias entre dados de teste vizinhos. A Situação 1, que sobrepõe dados de teste, é a que leva a um valor máximo de $SomaDist$, enquanto que na Situação 3 tem-se o menor valor. Portanto, usar $SomaDist$ como medida de diversidade levaria a uma situação indesejável de sobrepor dados de teste nos limites extremos do domínio de entrada. No caso de domínios

bidimensionais, por exemplo, a alocação que maximiza *SomaDist* para n dados de teste aloca $n/4$ dados de teste para cada canto do domínio.

Portanto, considerar a distância de cada dado de teste para o dado mais próximo (ao invés de considerar a distância para todos os dados) é coerente com o propósito de exercitar de forma completa o domínio de entrada. Pode-se dizer que, apesar de contra-intuitivo, maximizar *SomaDist* não leva ao aumento da diversidade, pelo menos do ponto de vista do teste de software.

No entanto, esta maneira de calcular a diversidade pode gerar uma situação indesejável de perda de informação. Considere a Figura 22, Situação 4 (os traços pontilhados representam distâncias entre os dados de teste vizinhos mais próximos uns dos outros). O dado t_2 é o mais próximo de t_1 , t_3 é o mais próximo de t_2 e vice versa, t_4 é mais próximo de t_5 e vice versa. Os dados de teste têm uma disposição tal que faz com que surjam dois subgrafos desconectados (t_1, t_2, t_3) e (t_4, t_5).

A situação 5 apresenta os mesmos subgrafos e, embora a localização dos dados seja diferente, o valor de diversidade será o mesmo em relação à situação 1. Intuitivamente o valor de diversidade deveria ser maior na situação 2 do que na situação 1 (veja discussão sobre continuidade de regiões de falha na próxima seção).

Uma alternativa para evitar esta perda de informação seria inserir uma ligação (ou ponte) entre subgrafos – representada pela linha tracejada na situação 5', neste caso, uma maior distância entre subgrafos seria identificada como um aumento da diversidade. Este recurso, entretanto, não se mostrou necessário. As meta-heurísticas para a geração dos conjuntos DOTS mostraram-se capazes de evoluir a diversidade mesmo desconsiderando a distância entre subgrafos. Aparentemente, os movimentos realizados pelas meta-heurísticas tendem naturalmente a afastar os dados de teste próximos e a conectar os subgrafos. Por exemplo, na situação 4, a tendência de movimentos seria afastar t_3 de t_2 , até uma situação em que o dado mais próximo de t_5 seja t_3 e não t_4 , conectando assim os dois subgrafos.

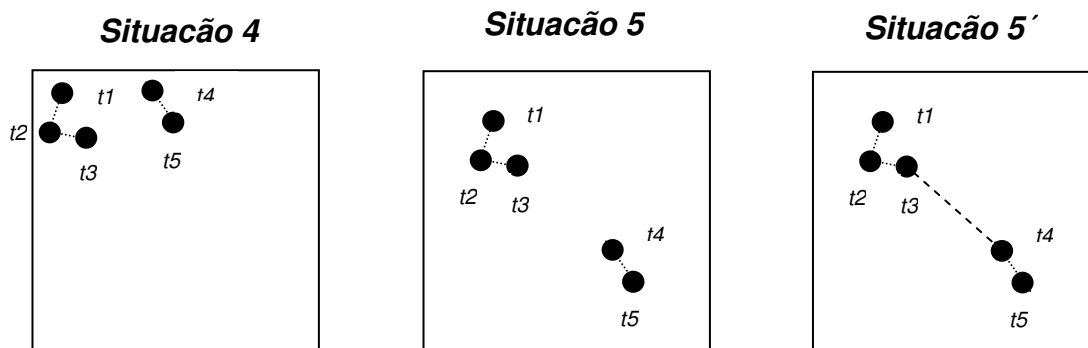


Figura 22. Perda de informação de diversidade

A Figura 23 mostra uma representação gráfica de dois conjuntos de dados de teste. Ambos os conjuntos são compostos por 100 dados de teste, cada um formado por duas variáveis de entrada do tipo real entre 0 e 100. Cada ponto representa um par de valores – um dado de teste. No lado esquerdo, tem-se um conjunto de dados de teste gerados aleatoriamente segundo uma distribuição uniforme, isto é, cada ponto do domínio tem a mesma chance de ser selecionado (*RTS*). Do lado direito, tem-se um conjunto de dados de teste orientado à diversidade (*DOTS*). Estes conjuntos são gerados automaticamente por meio de meta-heurísticas descritas na Seção 4.7 e apresentam altos níveis de diversidade. Este conjunto específico da Figura 23 foi gerado pelo algoritmo de Repulsão Simulada (SR-DOTG).

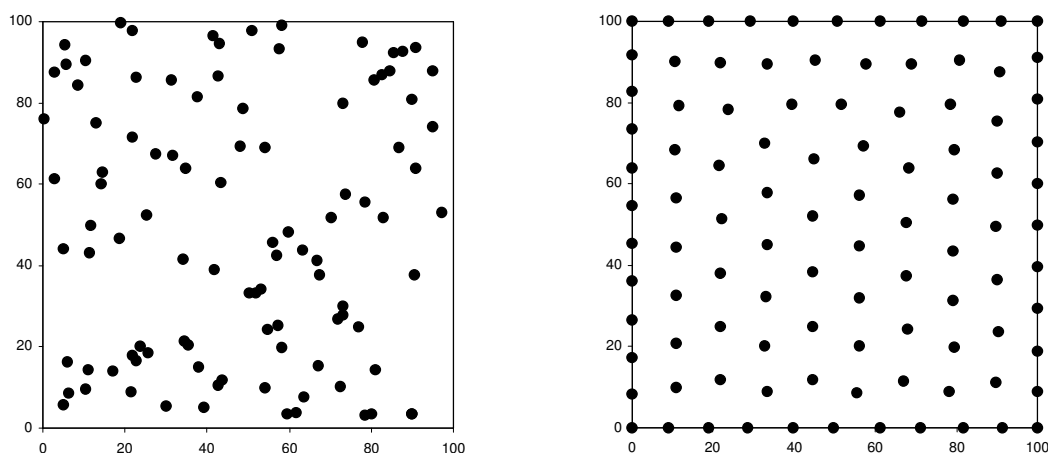


Figura 23. Conjunto RTS (esquerda) e conjunto DOTS-ID (direita)

Como pode-se perceber visualmente, os conjuntos *DOTS* apresentam as seguintes características:

- I. Os dados de teste cobrem uniformemente o domínio de entrada do programa, isto é, as distâncias entre cada dado de teste e seus vizinhos em D possuem baixa discrepância;
- II. Os pontos extremos do domínio de entrada do programa são cobertos por dados de teste; ou seja, os limites de D são exercitados;
- III. Para um certo D , ao se aumentar o tamanho do conjunto de dados de teste observa-se um aumento da densidade de dados de teste, que representa o número de dados de teste por unidade de área ou de “hiper-volume” de D . Dito de outro modo, conjuntos de dados de teste maiores levam a uma maior granularidade da cobertura de D por dados de teste.

O impacto dessas características para o teste de software é discutido nas seções seguintes.

4.4.3. Domínio de Busca e Domínio de Entrada

Um ponto importante é que esta medida de diversidade é computável diretamente para programas que recebam entradas numéricas (inteiros ou reais). Os algoritmos implementados para geração de *DOTS* também tratam dados de teste formados por variáveis numéricas.

Portanto, para a geração de dados, conforme detalhado no Capítulo 3, os algoritmos manipulam um Domínio de Busca associado ao Domínio de Entrada do Programa. O mapeamento do Domínio de Busca para o Domínio de Entrada do Programa pode ser direto (um para um). Neste caso, cada ponto no Domínio n -dimensional de Busca define um ponto no Domínio n -dimensional de Entrada do Programa; isto é, os valores das coordenadas do ponto no domínio de busca tornam-se os valores das variáveis de entrada dos dados de teste.

O teste de programas que recebem entradas mais complexas, envolvendo outros tipos (ex: caractere, lógico) ou estruturas, como vetores, matrizes ou registros, requer um mapeamento mais elaborado entre o domínio de busca e o domínio de entrada do programa. Este aspecto já foi descrito de forma genérica nas definições iniciais desta seção. Considerando especificamente a técnica *DOTG* e os algoritmos de geração de dados implementados, são necessárias considerações adicionais sobre esta questão.

São definidos os mapeamentos para estender a aplicação da técnica de teste proposta para programas que recebem entradas não numéricas. Estes mapeamentos podem ser agrupados como: Mapeamentos de Tipo e Mapeamentos de Estrutura.

Mapeamentos de Tipo: visam realizar a conversão de um valor numérico para um tipo não numérico. Exemplos:

- Mapeamento de valores numéricos para variáveis lógicas: um domínio numérico (exemplo: inteiro) é definido e dividido em duas seções (exemplo: 1 a 100 e 101 a 200); cada seção é associada a um valor lógico (exemplo: 1 a 100 associado a 0 e 101 a 200 associado a 1).
- Mapeamento de valores numéricos para caracteres: de maneira análoga à anterior, os caracteres podem ser tratados pela divisão de seções para representar os caracteres desejados. Um mapeamento um para um também pode ser feito, por exemplo, usando valores ASCII dos caracteres.
- Mapeamentos de valores numéricos para elementos de conjunto: para programas que recebam como entrada valores pertencentes a um conjunto, o mapeamento se dará pela definição de um vetor composto de uma estrutura contendo um campo *chave* e um campo *valor*. O campo *chave* é do tipo inteiro, enquanto que o campo *valor* contém um dos elementos do conjunto. Por exemplo, para gerar dados para um programa que receba como entrada um mês do ano, um vetor de 12 posições composto de {(1,"Jan"),(2,"Fev"), ... , (12,"Dez")} seria utilizado para realizar o mapeamento entre inteiros e *strings*.

Mapeamentos de Estrutura: visam realizar a conversão de variáveis primitivas para estruturas complexas. Exemplos:

- Mapeamento de reais ou inteiros para vetores ou matrizes: uma alternativa para este tipo de mapeamento consiste em tratar cada elemento do vetor ou matriz como uma variável distinta. Este tipo de tratamento é descrito nas definições iniciais desta seção. Deve-se notar que, no caso de uma matriz com N elementos, o domínio de busca será um espaço com N dimensões; cada ponto deste domínio representa um conjunto de N valores, um para cada elemento da matriz. Isto é, cada dado de teste (ponto no domínio de busca) é uma matriz, e o conjunto de dados de teste é um conjunto de matrizes.
- Mapeamento de reais ou inteiros para registros: cada elemento do registro é tratado como uma variável distinta. Por exemplo, se um programa tem como interface um registro com três campos, cada campo define uma dimensão no domínio de busca; portanto, cada ponto no domínio tridimensional de busca define um registro (dado de teste) e o conjunto de registros é o conjunto de dados de teste. Eventualmente, para registros com campos não numéricos, algum *Mapeamento de Tipo* pode ser necessário.

Os mapeamentos descritos são implementados por meio de módulos (*drivers*) que recebem dados numéricos dos algoritmos de geração de dados, realizam as conversões e atribuições necessárias, e chamam o programa em teste passando como parâmetros os dados de entrada segundo tipos e sequências definidas na interface do programa.

Outras transformações podem também ser realizadas nesses *drivers*, por exemplo: multiplicar um valor recebido por um fator e passar o novo valor para o programa em teste; receber dois valores do algoritmo de geração de dados, utilizá-los para computar o resultado de uma função e passar este resultado para o programa em teste, etc.

Os mapeamentos definidos não se propõem a serem completos, no sentido de permitir a utilização da técnica *DOTG* para o teste de qualquer programa, independentemente da sua interface. O objetivo de definir tais mapeamentos é fornecer direções para a utilização da

técnica de forma coerente em um leque maior de situações. Exemplos de aspectos não tratados neste trabalho incluem programas que recebem como entrada: estruturas dinâmicas como listas e árvores ou estruturas de dados de tamanho variável (definido dinamicamente); programas que recebem como entrada eventos situados no tempo; ou sequências de eventos e dados cuja ordem e/ou tamanho variem (sejam redefinidas) em cada execução.

4.5. Sumário e Propriedades de DOTS Para o Teste

A uniformidade alcançada na cobertura do domínio de entrada quando se utilizam os conjuntos de dados de teste orientados à diversidade (DOTS) influencia na eficácia do teste – neste contexto eficácia significa a chance de que o software falhe durante o teste.

Características dos DOTS permitem traçar hipóteses sobre as propriedades desses conjuntos para o teste de software:

- i. *DOTS* atingem alocam dados de teste para exercitar os limites do domínio de entrada do programa.
- ii. *DOTS* apresentam distâncias entre os pontos de teste bastante uniformes; isto permite um controle mais preciso da granularidade do conjunto de dados de teste. Variações no tamanho do conjunto de dados de teste refletem indiretamente na granularidade com que este conjunto cobre o domínio de entrada do programa.
- iii. Como as distâncias entre dados de teste adjacentes são similares, e controláveis (indiretamente) por meio da variação do tamanho do conjunto de dados de teste ($|T|$). O Valor $dist_test = Div(T) / |T|$, onde $Div(T)$ é a diversidade do conjunto de dados de teste T , representa o tamanho do espaço existente (em termos de distância euclidiana) entre dados de teste adjacentes.
- iv. Considerando regiões de falha que apresentem a característica de continuidade, isto é, que ocupem uma região contínua, como os “*Blob Defects*” (Bishop, 1993), ou os defeitos tipo faixa e tipo bloco (Chen et al., 2005), então haverá um determinado tamanho de região de falha $dist_falha$ em termos de distância euclidiana de cada

dimensão da região de falha. Para rever o conceito de *Região de Falha*, favor consultar o Capítulo 2, Seção 2.7.

- v. Se o tamanho do conjunto de dados de teste *DOTS* ($|T|$) for suficientemente grande para que a distância entre dados de teste adjacentes seja menor do que a o tamanho da região de falha ($dist_test < dist_falha$) então há uma grande chance de que algum dado de teste situe-se dentro da região de falha.

A Figura 24 mostra conjuntos de dados de teste *RTS* e *DOTS*, ambos com 100 dados de teste ($|T|=100$) – estes conjuntos são idênticos aos mostrados na Figura 18. Foram alocadas sobre esses conjuntos cinco regiões de falha idênticas do tipo Bloco. Este exemplo ilustra a característica de conjuntos *DOTS* de formarem uma espécie de “crivo” ou “trama” de dados de teste que torna difícil que as regiões de falha escapem de serem reveladas, desde que o valor $|T|$ seja grande o suficiente para que $dist_test$ tenha valor menor do que $dist_falha$. O conjunto *RTS* apresenta grandes espaços “vazios”, sem dados de teste, o que cria a possibilidade de que as regiões de falha permaneçam não descobertas. Conforme destacado no item (i) os conjuntos *DOTS* exercitam os limites do domínio de forma mais sistemática que os conjuntos *RTS*, esta característica é consonante com a observação de Beizer (1990) de que “os defeitos encontram-se nos cantos e reúnem-se nas fronteiras” (“*Bugs lurk in corners and congregate at boundaries*”). Além de uma quota bastante citada, esta incidência de defeitos relacionados ao comportamento do programa em situações limite é confirmada por praticantes de teste, tornando o teste dessas situações recomendado como uma “boa prática”.

Estas características tendem a fazer com que os conjuntos *DOTS* em média sejam mais eficazes do que os conjuntos *RTS*. O Capítulo 5 mostra os procedimentos realizados para a avaliação da técnica *DOTG*, os resultados obtidos sobre eficácia e a análise dos resultados.

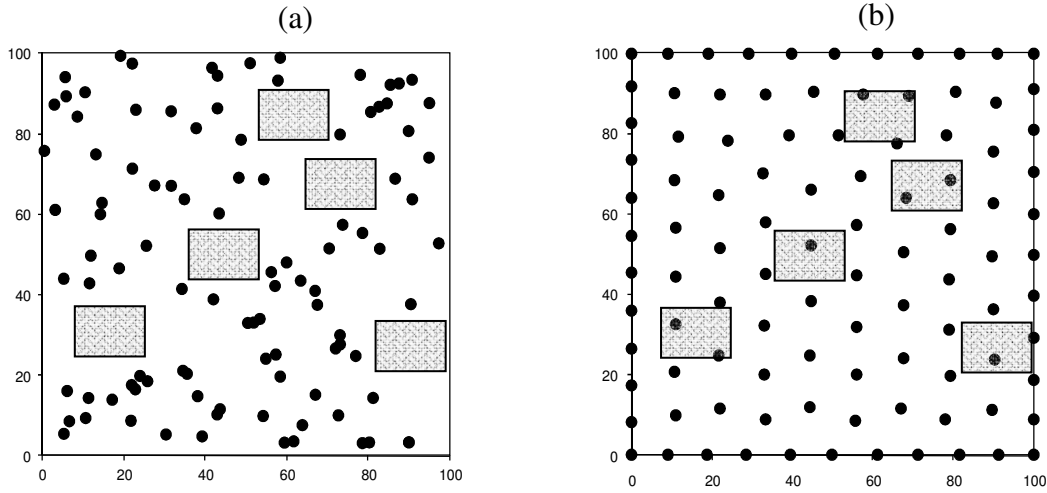


Figura 24. Regiões de falha e conjuntos RTS (a) e DOTS (b)

Um caso particular ilustra características da técnica. Dado um programa P com domínio de entrada D e um conjunto de dados de teste T de tamanho m . Considerando $m = |D|$ – o número de dados de teste é igual ao número de pontos no domínio de entrada – e supondo ainda existir apenas um ponto de D que provoque falha isto é, a taxa de falhas é $\theta = 1/m$.

Neste caso o único conjunto de dados de teste com diversidade máxima é o que aloca cada dado de teste em um ponto diferente de D , sem sobrepor nenhum dado de teste. Trata-se obviamente da alocação ótima dos dados, facilmente identificada por um testador humano: alocar dois dados em um mesmo local não faz sentido, visto que os dois teriam certamente o mesmo resultado ao serem executados (ambos provocam falha ou ambos não provocam falha). Este conjunto de dados de teste com diversidade máxima é o único que garante a ocorrência da falha (chance de falha $f = 1$). À medida que a diversidade de T ($Div(T)$) diminui, mais dados de teste ficam sobrepostos e uma maior porção do domínio fica sem testes. Portanto, à medida que $Div(T)$ diminui, diminui também a chance de ocorrência da falha. Esta chance será mínima quando todos os dados de teste estão sobrepostos em algum ponto de D . Neste caso tem-se $Div(T) = 0$ e chance de falha $= 1 / |D|$.

Conforme destacado, trata-se de um caso particular. Em geral o domínio de entrada do programa é expressivamente maior do que o número de dados de teste ($|D| \gg m$).

Portanto na maioria das situações é esperado que pontos do domínio de entrada não sejam testados; nestes casos os conjuntos *DOTS* combinam faixas de valores para as variáveis de entrada de modo a cobrir de forma completa o domínio de entrada, evitando grandes “vazios” sem dados de teste.

4.6. Meta-heurísticas Para a Geração de Conjuntos de Teste Orientados à Diversidade (DOTS)

O problema de gerar conjuntos de dados de teste orientados à diversidade na perspectiva do domínio de entrada é similar aos problemas Np-completos de embalar esferas e de dispersar pontos. O problema embalar esferas consiste em posicionar esferas em um volume de forma que elas não se sobreponham umas às outras e que a soma dos volumes das esferas embaladas é maximizado (Hales, 1992), (Conway et al., 1999). O problema de dispersar pontos consiste em posicionar n pontos em um quadrado de forma que a distância mínima entre os pontos é maximizada (Goldberg, 1970), (Fujisawa e Takefuji, 1995).

Devido a esta complexidade, decidiu-se abordar o problema prático de gerar conjuntos de dados de teste orientados à diversidade (DOTS) por meio de técnicas meta-heurísticas. Conforme já destacado no início deste capítulo, foram implementadas três meta-heurísticas: Recozimento Simulado – RS (Simulated Annealing); Algoritmo Genético – AG (Genetic Algorithm) e a Repulsão Simulada (Simulated Repulsion – SR). As meta-heurísticas RS e AG são amplamente conhecidas e foram descritas no Capítulo 3a a meta-heurística Repulsão Simulada (SR), proposta nesta tese, é baseada na dinâmica de sistemas de partículas.

4.6.1. Recozimento Simulado Para Geração de DOTS

Esta implementação é baseada no algoritmo Recozimento Simulado para Diversidade (SAND), proposto como uma abordagem imunológica para iniciar pesos em redes neurais (De Castro, Von Zuben, 2001c).

O algoritmo denominado SA-DOTG inicia com um conjunto de dados de teste gerado aleatoriamente, composto por um número específico de dados de teste, sendo que cada dado de teste é formado por variáveis de entrada. Um novo conjunto de dados de teste é gerado pela realização de mudanças aleatórias (mutações). Uma mutação é realizada selecionando aleatoriamente variáveis de entrada do conjunto de dados de teste com uma probabilidade de mutação mp e trocando os valores por outros gerados aleatoriamente. Os novos valores são gerados considerando tipo da variável de entrada e a faixa de valores permitida.

A variação de diversidade entre o conjunto de dados de teste alterado e o conjunto original é calculada pela aplicação da Equação 4.2 para cada conjunto. Se a diferença no valor de diversidade é positiva (diversidade do conjunto de dados alterado maior que o original) o conjunto de dados de teste alterado torna-se a solução corrente para a próxima iteração do SA-DOTG. Conjuntos de dados alterados com diversidade menor que a do conjunto original também podem ser aceitos como nova solução corrente. Esta aceitação é controlada por uma função que usa números aleatórios, a variação da diversidade e o parâmetro temperatura (tp). Conjuntos de dados alterados “piores” (com menor diversidade) são aceitos com a probabilidade controlada (Função 4.4):

$$random[0,1) < e^{\frac{Div(New_test_set) - Div(Test_set)}{tp}} \quad (4.4)$$

Um cronograma de resfriamento realiza uma redução do valor de tp ao longo da busca e influencia como mudanças nos conjuntos de teste são aceitas em cada fase da busca. O SA-DOTG reduz o valor de tp a cada ks iterações usando $tp = tp \cdot \alpha$. Os valores para os parâmetros temperatura inicial ($tini$), para ks e para α foram definidos executando

repetidamente o algoritmo e observando o desempenho em termos de capacidade de otimizar a diversidade ao longo das iterações. Valores tipicamente usados: $\alpha = 0.95$; $ks = 100$; $tini = 1000$.

A busca ocorre até que um valor limite inferior para tp seja atingido, ou até que nenhum aumento da diversidade (valor Div) seja observado em um número de iterações.

O SA-DOTG recebe como entrada os parâmetros: número de dados de teste (NTD); número de variáveis de entrada (NIV); a descrição de cada variável de entrada do dado de teste: tipo da variável, limite inferior de valores, limite superior de valores; e os parâmetros de busca ($tini$, ks , mp). Ao final da busca, o conjunto de teste de alta diversidade (DOTS) é fornecido ao testador.

A Figura 25 detalha o SA-DOTG. Linha 1 – SA-DOTG recebe parâmetros número de dados de teste (NTD) e número de variáveis de entrada (NIV). Linha 3 – receber a descrição de cada variável de entrada do dado de teste: tipo, limites inferiores e superiores de valores. Linha 4 – aceita valores para: a temperatura inicial ($tini$), o limite inferior de temperatura ($lowerTp$), o número de iterações para cada diminuição de temperatura (ks), e para a probabilidade de mutação (mp). Linha 7 – valores para as variáveis de entrada de cada dado de teste são gerados aleatoriamente. O SA-DOTG entra em dois laços que controlam as iterações e a diminuição do valor de temperatura (Tp). Em cada iteração, um novo conjunto de teste é gerado pela aplicação da operação de mutação no conjunto de teste atual, com probabilidade controlada – linha 14. A variação de diversidade é computada – linha 15. Se a diversidade aumenta o novo conjunto de teste é aceito incondicionalmente para a próxima iteração – linha 17; caso contrário, é realizada a aceitação condicional baseada na Função 4.4 – linha 19. Ao final da busca o conjunto DOTS é fornecido como saída ao testador.

```

begin_SA
/** input parameters **/
/* NTD: number of test data in the test set */
/* NIV: number of input variables in the test data */
/* type, inf_limit, sup_limit: type, inferior and superior values for each input variable */
/* tini, lowerTp: initial temperature, lower bound for temperature */
/* ks, mp number of interactions for temperature reductions; mutation probability */
/** functions **/
/* uniform_random: returns a random value uniformly generated between specified limits and of
given type */
/* mutate: returns a new test set with a randomly generated perturbation */
/* evaluate: returns the test set diversity value */
/* rand(0,1): returns a random number between 0 and 1 */
1. read (NTD,NIV);
2. for ( iv = 1 to NIV )
3.   read(type[iv], inf_limit[iv], sup_limit[iv]);
4. read (tini, lowerTp, ks, mp);
5. for t=1 to NTD
6.   for x=1 to NIV
7.     test_set[t,x] = uniform_random(type[x], inf_limit[x], sup_limit[x]);
8.   endfor
9. endfor
10. tp = tini;
11. while (tp > LowerTp)
12.   rc=1;
13.   while (rc < Ks)
14.     new_test_set = mutate(test_set);
15.     deltadiv = evaluate(new_test_set) – evaluate(test_set);
16.     if (deltadiv > 0)
17.       test_set = new_test_set;
18.     else
19.       if ( rand(0,1) < exp(deltadiv / tp) )
20.         test_set = new_test_set;
21.       rc = rc+1;
22.   endwhile
23.   tp = tp *  $\alpha$ ;
24. endwhile
25. write (test_set);
26. end_SA

```

Figura 25. SA-DOTG

4.6.2. Algoritmo Genético para Geração de DOTS

O Algoritmo Genético (denominado GA-DOTG) trabalha com uma população inicial de conjuntos de teste gerados aleatoriamente; cada conjunto de teste é composto por um número específico de dados de teste e cada dado de teste é formado pelas variáveis de entrada definidas. O GA-DOTG seleciona iterativamente, a cada geração, conjuntos de teste para a geração seguinte, associando a cada conjunto de teste uma chance de seleção

baseada na diversidade do conjunto (calculada usando a Equação 4.2). Os conjuntos de teste selecionados são submetidos à operação de recombinação (troca de valores associados às variáveis de dados de teste dos pares de conjuntos de teste) e a mutação (mudanças aleatórias de valores para variáveis de entrada). A seleção e a recombinação de conjuntos de teste, realizadas repetidamente, movem a busca em direção a conjuntos de teste de alta diversidade.

Seleção, Recombinação e Mutação

O GA-DOTG aplica operadores genéticos padrões, bem aceitos. A cada geração a seleção proporcional é aplicada para escolher os melhores indivíduos (conjuntos de teste) para a geração seguinte, a recombinação uniforme é utilizada para trocar material genético entre os indivíduos, e a mutação simples introduz novos valores para os genes (variáveis de entrada) selecionados (Goldberg, 1989), (Michalewicz, 1996).

O valor de objetivo (fitness) para cada conjunto de teste é o seu valor de diversidade, calculado usando a Equação 4.2. A seleção proporcional imita o fenômeno natural da sobrevivência do mais ajustado. Um conjunto de teste T_i é selecionado para a próxima geração usando uma “roleta” (*roulette wheel*) que faz com que a chance de um conjunto de teste ser selecionado seja proporcional a razão entre o seu valor objetivo ($Div(T_i)$) e a média dos valores objetivo de toda a população de conjuntos de teste ($AvgDiv$) – Equação 4.5. O GA-DOTG também utiliza o modelo elitista, mantendo sempre o melhor conjunto de teste da população atual na próxima geração (sem nenhuma alteração).

$$P(T_i) = \frac{Div(T_i)}{AvgDiv} \quad (4.5)$$

A recombinação uniforme é realizada pela troca de valores para as variáveis de entrada entre pares de conjuntos de teste. Cada variável de entrada, de cada dado de teste, cujo conjunto forma o conjunto de teste é considerada independentemente, e é trocada com uma probabilidade pc . O operador de mutação é responsável pela introdução de novos

valores para as variáveis de entrada na população de conjuntos de teste. Valores originais das variáveis de entrada são alterados para novos valores gerados aleatoriamente, de acordo com o tipo e domínio da variável, com uma probabilidade p_m independente para cada variável.

O GA-DOTG recebe os parâmetros: número de conjuntos de teste (NTS); número de dados de teste de cada conjunto (NTD); número de variáveis de entrada de cada dado de teste (NIV); descrição de cada variável de entrada: tipo, limites superior e inferior de valores; e parâmetros de busca (p_m , p_c , e o limite superior para o número de gerações).

Valores para as variáveis de entrada são gerados aleatoriamente para cada variável, cada dado de teste e cada conjunto de teste. Os conjuntos de teste gerados aleatoriamente são avaliados por meio do cálculo dos valores de diversidade de cada conjunto de teste. Os conjuntos de teste são então selecionados iterativamente (a chance de seleção é proporcional ao valor de diversidade) e modificados com a aplicação de operadores genéticos. Ao final da busca os conjuntos de alta diversidade (DOTSs) são fornecidos como saída ao testador.

A Figura 26 detalha o GA-DOTG. Linha 1 – o GA-DOTG recebe os parâmetros: número de conjuntos de teste (NTS); número de dados de teste (NTD); número de variáveis de entrada (NIV). Linha 3 – recebe a descrição de cada variável de entrada: tipo, limites superior e inferior de valores. Linha 4 – recebe os parâmetros probabilidade de mutação, probabilidade de recombinação e limite superior para o número de iterações. Linha 9 – valores para as variáveis de entrada são gerados aleatoriamente para cada variável, dado de teste, e conjunto de teste. Os conjuntos de teste gerados aleatoriamente são avaliados – linha 12.

O laço da linha 14 à linha 28 controla a sucessão de gerações do GA-DOTG até um limite, quando os conjuntos de teste são fornecidos ao testador. O laço interno da linha 16 à linha 28 realiza a seleção e aplica os operadores genéticos na população atual de conjuntos de teste. Em cada iteração do laço dois conjuntos de teste são selecionados pela função *seleciona*, que retorna os dois conjuntos de teste escolhidos – linhas 18 e 19. A chance de um conjunto de teste ser selecionado é proporcional ao valor de diversidade do conjunto. A

operação de recombinação realiza a troca de valores para as variáveis de entrada dos conjuntos de teste selecionados, gerando um novo par de conjuntos de teste – linha 20. O operador mutação é aplicado, resultando em dois novos conjuntos de teste com valores para as variáveis de entrada alterados. Esses novos conjuntos de teste têm os valores de diversidade calculados pela função evaluate – linhas 23 e 24.

```

begin_GA
/** input parameters **/
    /* NTS: number of test sets in the population */
    /* NTD: number of test data in each test set */
    /* NIV: number of input variables in the test data */
    /* type, inf_limit, sup_limit: type, inferior and superior values for each input variable */
    /* pm, pc, iter_limit: mutation probability, crossover probability, maximum number of iterations*/
/** functions **/
    /* uniform_random: returns a random value uniformly generated between specified limits and of
    given type */
    /* select: returns a test set chosen by using the proportional selection */
    /* crossover: applies the uniform crossover on two test sets – mate_1 and mate_2 – generating two
    new test */    /* sets. pc controls the probability of exchanging each input variable */
    /* mutate: creates a new test set with the values for input variables changed with probability pm */
    /* evaluate: returns the test set diversity value */
1.  read (NTS,NTD,NIV);
2.  for ( iv = 1 to NIV )
3.      read(type[iv], inf_limit[iv], sup_limit[iv]);
4.  read (pm, pc, iter_limit);
5.  gen = 1;
6.  for T=1 to NTS
7.      for t=1 to NTD
8.          for x=1 to NIV
9.              new_test_set[T,t,x] = uniform_random(type[x], inf_limit[x], sup_limit[x]);
10.         endfor
11.     endfor
12.     div[T] = evaluate(new_test_set[T]);
13. endfor
14. while (gen < iter_limit)
15.     T=1;
16.     for i=1 to NTS test_set[i] = new_test_set[i];
17.     while (T <= NTS)
18.         mate_1 = select (test_set, div);
19.         mate_2 = select (test_set, div);
20.         crossover(test_set[mate_1],test_set[mate_2], new_test_set[T], new_test_set[T+1],pc);
21.         new_test_set[T] = mutate(new_test_set[T], pm);
22.         new_test_set[T+1] = mutate(new_test_set[T+1], pm);
23.         div[T] = evaluate(new_test_set[T]);
24.         div[T+1] = evaluate(new_test_set[T+1]);
25.         T= T+2;
26.     endwhile
27.     gen=gen+1;
28. endwhile
29. for i=1 to NTS write(test_set[i]);
30. end_GA

```

Figura 26. GA-DOTG

4.6.3. Repulsão Simulada: uma Meta-heurística Auto-organizável para Otimizar Diversidade

As meta-heurísticas SA-DOTG e GA-DOTG são baseadas em abordagens amplamente utilizadas para otimização, o Recozimento Simulado e os Algoritmos Genéticos, respectivamente. RS e AG são ferramentas robustas utilizáveis em situações genéricas, independentemente de características da função a ser otimizada. Deste modo SA-DOTG e GA-DOTG são aplicáveis potencialmente para a geração de conjuntos de teste de alta diversidade (DOTS) para qualquer perspectiva, seja utilizando informações sobre as entradas recebidas pelo software (perspectiva do domínio de entrada – DOTG-ID) ou informações de outras naturezas, tais como requisitos funcionais do software, modelos de comportamento, código fonte, etc.

A meta-heurística Repulsão Simulada para geração de dados de teste (chamada SR-DOTG), proposta nesta tese, foi definida com o propósito de gerar conjuntos de teste de alta diversidade na perspectiva do domínio de entrada – DOTG-ID. A SR-DOTG é uma abordagem específica para geração de conjuntos DOTG-ID e apresenta um mecanismo que leva os dados de teste a “espontaneamente” se posicionarem no domínio de entrada do programa de forma a otimizar a diversidade do conjunto de teste. Conforme destacam (Trojanowski, Michalewicz, 1997) “considerar conhecimento específico do problema pode levar a soluções mais eficientes e eficazes para a solução do problema” (Trojanowski, Michalewicz, 1997), ponto destacado também em (Droste e Wiesmann, 2003).

Enquanto as meta-heurísticas SA-DOTG e GA-DOTG utilizam a medida de diversidade para direcionar a busca por conjuntos de teste de alta diversidade (DOTS), a SR-DOTG tem o seu funcionamento baseado em um comportamento emergente de auto-organização dos dados de teste no domínio de entrada do programa. Este comportamento ocorre por meio da simulação de sistemas de partículas (Witkin, 1997), (Hockney e Eastwood, 1988).

Visão Geral da SR-DOTG

A otimização baseada em enxame de partículas, abordada no Capítulo 3, simula comportamentos típicos de enxames, como revoadas de pássaros, ou cardumes de peixes (Kennedy e Eberhart, 1995), (Kennedy e Eberhart, 2001). (Eberhart e Shi, 2001). A SR-DOTG, no entanto, não aplica esta metáfora de comportamento social, mas a metáfora de um comportamento físico, simulando forças e movimentos originados por um fenômeno magnético em partículas. Por causa das características do sistema de partículas, cada dado de teste, representado como uma partícula do sistema, espontaneamente se posiciona no domínio de entrada do programa de maneira tal que a diversidade do conjunto de teste seja maximizada.

Cada dado de teste recebe uma carga elétrica positiva (Q), cujo valor inicial é um parâmetro da SR-DOTG. Estas cargas fazem com que cada dado de teste crie um campo elétrico ao seu redor. Esses campos em conjunto resultam em forças aplicadas nos dados de teste que causam o movimento desses dados no domínio de entrada do programa. Uma descrição em alto nível da SR-DOTG é apresentada na Figura 27. Sistemas de partículas (conjuntos de teste) são gerados, as forças de interação entre as partículas são calculadas e a diversidade de cada sistema é avaliada. Tem-se então um laço que controla cada iteração, nas quais as posições de cada partícula são atualizadas, as forças de interação são recalculadas, e a diversidade de cada sistema é reavaliada.

```
Repulsão Simulada
{
  iniciar sistemas de partículas;
  calcular forças de interação;
  avaliar diversidade;
  enquanto critério de término não atingido
  {
    movimento: atualizar posições das partículas;
    calcular forças de interação;
    avaliar diversidade;
  }
}
```

Figura 27. Meta-heurística Repulsão Simulada

A SR-DOTG trata *NTS* conjuntos de teste $T (T_1, T_2, \dots, T_{NTS})$. Cada conjunto de teste contém *NTD* dados de teste $t (t_1, t_2, \dots, t_{NTD})$. Cada dado de teste contém *NIV* variáveis de entrada $x (x_1, x_2, \dots, x_{NIV})$. Os valores *NTS*, *NTD*, *NIV* e a descrição de cada variável de entrada x são parâmetros de entrada da meta-heurística. A descrição de cada variável de entrada x inclui o seu tipo (inteiro ou real) e a faixa de valores que podem ser atribuídos à variável (por exemplo, inteiro entre 0 e 5000). Assume-se que essas faixas de valores sejam sempre contínuas.

Um gerador de números aleatórios uniformes é utilizado para gerar os *NTS* conjuntos iniciais de teste. Cada dado de teste de cada conjunto de teste representa uma partícula localizada no domínio de entrada do programa (D), cujo número de dimensões é dado pelo número de variáveis de entrada (*NIV*). O valor de cada variável de entrada do dado de teste define a posição deste dado em relação a cada dimensão de D . Cada conjunto de teste T é um sistema de partículas que evolui de forma independente e concorrente com os outros conjuntos de teste. Ou seja, a posição dos dados de teste de um conjunto T não interfere nas posições dos dados de teste dos outros conjuntos.

Optou-se por gerar de forma concorrente mais de um conjunto de teste para facilitar as avaliações empíricas realizadas. A SR-DOTG pode também ser empregada para gerar apenas um conjunto de teste, em situações normais de uso. Neste caso, ganha-se em desempenho, pois a geração de apenas um conjunto de teste é, obviamente, mais rápida do que a geração de múltiplos conjuntos.

Forças e Movimentos de Dados de Teste

A cada iteração da SR-DOTG todos os *NTS* conjuntos de teste têm as posições de seus dados de teste atualizadas de acordo com as mesmas leis físicas. As forças entre dois dados de teste são definidas por meio de uma adaptação da Lei de Coulomb – Volume 3 (Eletromagnetismo) de Halliday et al (2009): a magnitude da força eletrostática entre duas cargas elétricas pontuais é diretamente proporcional ao produto da magnitude de cada carga e inversamente proporcional ao quadrado da distância entre estas cargas (Equação 4.6).

$$\vec{F}(t_i, t_j) = Q^2 / d(t_i, t_j)^2 \quad (4.6)$$

Na qual Q é o valor atual de carga dos dados de teste e d é a distância euclidiana entre t_i e t_j . A força resultante RF em um dado de teste t_i é a soma vetorial das forças induzidas em t_i por todos os outros t_j dados de teste de T ($1 \leq j \leq NTD, i \neq j$) (Equação 4.7). À medida que as iterações se sucedem, o valor de Q é incrementalmente reduzido (por um “coeficiente de descarga”), que faz com que as forças resultantes sejam progressivamente menores, assim como os movimentos resultantes dos dados de teste.

$$\overrightarrow{RF}(t_i) = \sum_{j=1(j \neq i)}^{NTD} F(t_i, t_j) \quad (4.7)$$

A Figura 28 (esquerda) ilustra o cálculo da força eletrostática F entre dois dados de teste entre t_i e t_j . A Figura 28 (direita) ilustra o cálculo da força em t_3 resultante da interação de t_1 e t_2 com t_3 (soma vetorial de valores F): $RF(t_3) = F(t_3, t_1) + F(t_3, t_2)$.

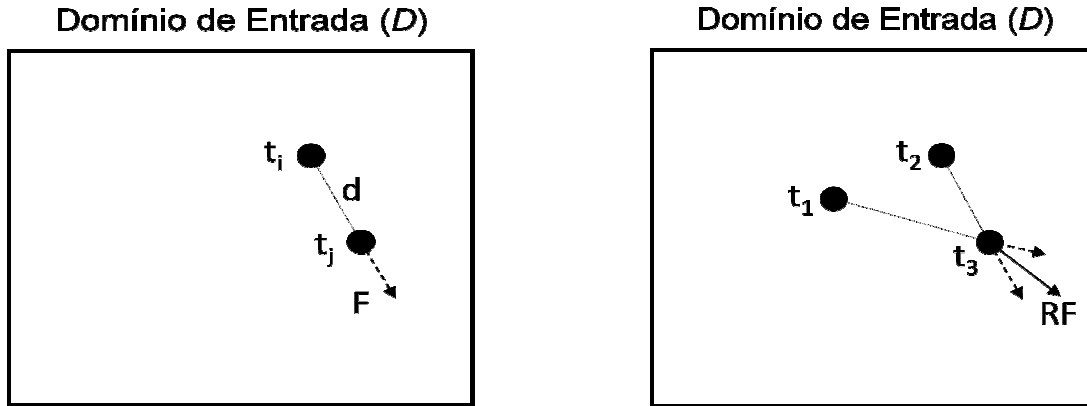


Figura 28. Cálculo de forças eletrostáticas entre dados de teste

De uma iteração para a próxima, todos os dados de teste de todos os conjuntos de teste têm as suas posições atualizadas. A nova posição de um dado de teste é definida usando uma adaptação das leis de mecânicas de Newton (Equação 4.8).

$$t_{i,new} = t_i + (\overrightarrow{RF}(t_i)/m) \quad (4.8)$$

Na qual $t_{i\text{new}}$ é a nova posição do dado de teste t_i , $RF(t_i)$ é a força resultante induzida em t_i pelos outros dados de teste do conjunto, e m é um valor de massa constante para todos os dados de teste. A massa do dado de teste (um valor numérico positivo, exemplo: $m = 10$) visa simplesmente restringir os movimentos dos dados de teste que tenderiam a deslocamentos infinitos com valor $m = 0$. A Figura 29 (esquerda) ilustra o deslocamento de t_3 para uma nova posição $t_{3\text{new}}$.

Se a nova posição do dado de teste após o movimento ($t_{i\text{new}}$) está localizada fora do domínio de entrada (D) é simulada uma colisão elástica entre o dado de teste e os limites de D . Neste caso, a dimensão específica (ou dimensões) em que $t_{i\text{new}}$ estaria localizado fora de D funciona como uma barreira física na qual o dado de teste colide e retorna para dentro de D . A colisão em bordas de D é parcialmente elástica, isto é a colisão dissipa energia. Após a colisão, a nova posição do dado de teste é mais próxima relação à borda onde ocorreu a colisão do que era a posição antes ao movimento. O coeficiente elástico é um parâmetro ajustado em torno de 0.9. A Figura 29 (direita) ilustra a situação em que um dado de teste seria posicionado fora do domínio de entrada do programa (posição $t_{i\text{new}}$). A colisão simulada reposiciona o dado de teste em D (posição $t_{i\text{new}}$).

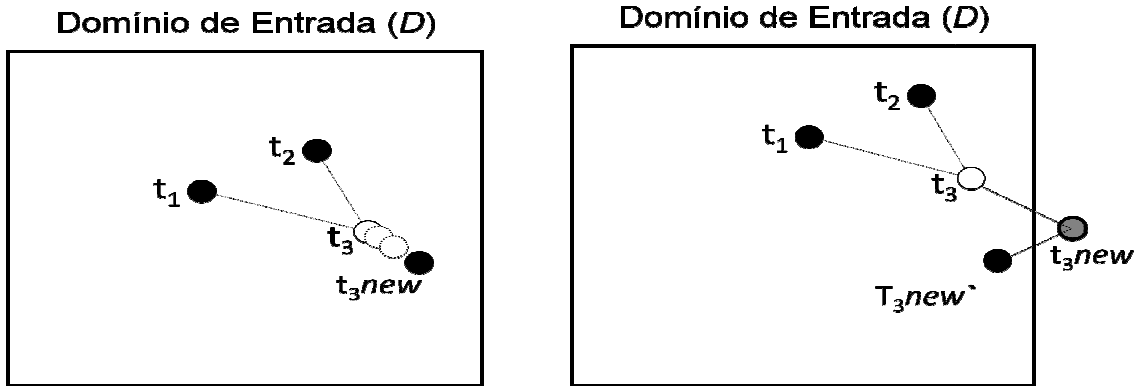


Figura 29. Movimentos de dados de teste

Imagens instantâneas (*snapshots*) dos conjuntos de teste em cada iteração ao longo da otimização realizada pela SR-DOTG mostram um padrão de comportamento. Dados de teste localizados em posições mais centrais no domínio de entrada estão sujeitos a forças resultantes mais baixas e se movem mais lentamente. Dados de teste situados próximos das

bordas do domínio de entrada são lançados em direção a esses limites do domínio. A colisão elástica ocorre de forma que os dados de teste “repiquem” nessas bordas, acomodando-se à medida que as iterações se sucedem.

Com o progresso da busca as forças resultantes nos dados de teste tendem a reduzir, levando também a uma diminuição nos movimentos dos dados de teste. Esta redução ocorre devido a dois fatores, pela redução do valor de carga dos dados de teste (Q) e pela “acomodação” dos dados de teste em posições nas quais as forças resultantes tendam a valores progressivamente menores. Nos estágios finais da busca os sistemas de partículas (conjuntos de teste) atingem uma situação de equilíbrio de forças, os movimentos tendem a cessar, e o valor de diversidade dos conjuntos de teste se estabiliza em patamares elevados.

Detalhamento da SR-DOTG

A Figura 30 detalha a SR-DOTG. Linha 1 – SR-DOTG recebe os parâmetros: número de conjuntos de teste; número de dados de teste; número de variáveis de entrada. Linha 3 – recebe a descrição de cada variável de entrada que forma o dado de teste: tipo, limites superior e inferior de valores. Linha 4 – recebe os parâmetros: valor inicial para carga dos dados de teste; limite superior para o número de iterações; e coeficiente de descarga. Linha 6 – é definida a massa de cada dado de teste; linha 10 – são gerados aleatoriamente valores para cada variável de entrada, de cada dado de teste e de cada conjunto de teste. O laço na linha 12 controla as iterações da SR-DOTG. Em cada iteração são calculadas as distâncias de cada dado de teste para todos os outros dados de teste – linhas 19 e 20. A força total aplicada no dado de teste é calculada – linha 20 e é decomposta com respeito a cada dimensão de D (cada dimensão relacionada a uma variável de entrada) – linha 22. A nova posição do dado de teste é calculada com respeito a cada dimensão de D – linha 25. É avaliado se o dado de teste está fora do domínio de entrada em relação a cada dimensão de D . Neste caso é simulada uma colisão elástica do dado de teste na borda específica do domínio D . A diversidade de cada conjunto de teste é calculada – linha 31, as cargas dos dados de teste são reduzidas – linha 33, e o contador de iterações é incrementado – linha 34. Na linha 35 os novos conjuntos de teste são tomados para a próxima iteração. Ao final da busca os vários conjuntos de teste de alta diversidade (DOTS) são fornecidos ao testador.

```

begin_SR
/** input parameters */
    /* NTS: number of test sets in the population */
    /* NTD: number of test data in each test set */
    /* NIV: number of input variables in the test data */
    /* type, inf_limit, sup_limit: type, inferior and superior values for each input variable */
    /* initial_charge, iter_limit, discharge_coef: initial value for charge, maximum number of iterations,
    discharge */ /* coefficient */
/** functions */
    /* uniform_random: returns a random value uniformly generated between specified limits and of
    given type */
    /* compute_distance: compute the distance between two test data with respect to a input variable */
    /* outside_domain: returns true if there is any test data out of the input domain with respect to a input
    variable */
    /* elastic_colision: simulates a elastic collision on the domain boundary with respect to a input
    variable */
    /* evaluate_report_diversity: compute and report the diversity value of a test set */
1.  read (NTS,NTD,NIV);
2.  for ( iv = 1 to NIV )
3.      read(type[iv], inf_limit[iv], sup_limit[iv]);
4.  read (initial_charge, iter_limit, discharge_coef));
5.  Q = initial_charge;
6.  data_mass = 10; /* data_mass can be fixed in a small value */
7.  for T=1 to NTS
8.      for t=1 to NTD
9.          for x=1 to NIV
10.             test_set[T,t,x] = uniform_random(type[x], inf_limit[x], sup_limit[x]);
11. iteration = 1;
12. while (iteration < iter_limit)
13.     for T=1 to NTS
14.         for t1=1 to NTD
15.             for t2=1 to NTD
16.                 for x=1 to NIV
17.                     distance[x] = compute_distance(test_set[T,t1,x], test_set[T,t2,x]);
18.                     distance = distance + (distance[x])2;
19.                 endfor(x)
20.                 force = Q2/distance;
21.                 for x=1 to NIV
22.                     force[x]=force[x]+force*(distance[x] / distance);
23.                 endfor(t2)
24.                 for x=1 to NIV
25.                     new_test_set[T,t1,x] = test_set[T,t1,x] + (force[x]/data_mass);
26.                     if (outside_domain(new_test_set[T,t1,x]))
27.                         new_test_set[T,t1,x] = elastic_colision(test_set[T,t1,x]);
28.                     endfor(x)
29.                     for i=1 to NIV force[i]=0;
30.                 endfor(t1)
31.                 evaluate_report_diversity(Test_set[T]);
32.             endfor(T)
33.             Q = Q * discharge_coef;
34.             iteration++;
35.         for T=1 to NTS test_set[T] = new_test_set[T];
36.     end while
37. for T=1 to NTS write(test_set[T]);
38. end_SR

```

Figura 30. Detalhamento da meta-heurística Repulsão Simulada

Algumas observações adicionais sobre a meta-heurística proposta são necessárias.

- Todas as regras de movimento são na verdade adaptações de leis da física. As regras de movimento são baseadas na lei de Newton: $a=f/m$; (a : aceleração, f : força, m : massa). Volume 1 (Mecânica) de (Halliday et al., 2009).

O objetivo não é criar movimentos realistas, mas sim fazer com que a disposição final de partículas (dados de teste) seja tal que maximize valores de diversidade. Portanto, buscou-se reduzir a complexidade da implementação e o custo computacional para gerar os conjuntos DOTS, em detrimento do realismo de movimentos.

- As novas posições dos dados de teste são calculadas considerando os valores de aceleração, ao invés de considerar aceleração e velocidade das partículas.
- Nas colisões de dados de teste com as bordas de D, as novas posições dos dados de teste são sempre mais próximas da borda do que antes da colisão.
- Colisões entre dados de teste não são tratadas.
- A definição de valores iniciais para a carga (Q) requer algum cuidado. Valores muito pequenos para Q fazem com que os movimentos dos dados de teste sejam muito lentos, o que é percebido como um aumento relativamente lento dos valores de diversidade ao longo das iterações. Por outro lado, valores muito altos para Q parecem causar uma instabilidade que faz com que a diversidade dos sistemas de partículas não aumente ao longo das interações. Na utilização da SR-DOTG foi percebido que existe um valor limite superior para Q a partir do qual ocorre a instabilidade (chamado Q_{limite}).

Embora não tenha sido possível associar precisamente Q_{limite} com outros parâmetros de busca, foi observado que para um domínio de entrada D, os valores Q_{limite} são inversamente proporcionais ao tamanho dos conjuntos de teste (ITl). Esta relação é plausível, visto que aumentar o número de dados de teste tende a aumentar as forças resultantes e os deslocamentos produzidos nos dados de teste. Valores de Q acima de Q_{limite} geram, portanto, um efeito de “saturação” e o

comportamento dos dados de teste falha em maximizar a diversidade dos conjuntos de teste. Quando ocorre, este fenômeno é facilmente percebido pelo monitoramento do progresso da busca.

A abordagem prática para definir Q é fazer analogia com valores utilizados em experiências anteriores, tomando $|T|$ como parâmetro de referência para ajustar Q . Desde que o valor Q esteja abaixo de Q_{limite} , o desempenho da SR-DOTG para maximizar a diversidade não é muito sensível a variações deste parâmetro.

- Um escalonamento de descarga diminui progressivamente o valor da carga dos dados de teste e as forças resultantes ao longo da busca. O coeficiente de descarga utilizado é em torno de 0.95. Isto leva a um ajuste fino dos movimentos nos estágios finais da busca. Embora esta redução progressiva de Q permita que se alcancem valores maiores de diversidade final dos conjuntos de teste, este aspecto não é determinante no funcionamento da meta-heurística. Mesmo com Q constante a SR-DOTG produz o comportamento desejado, gerando conjuntos de teste de alta diversidade como resultado.

4.7. Trabalhos Relacionados

Esta seção descreve trabalhos que apresentam algum aspecto em comum com a técnica DOTG. Para cada trabalho, são identificados os pontos em comum e as diferenças da proposta apresentada em relação à técnica DOTG.

É descrita a Seleção de Dados de Teste por Análise e Clusterização (Leon e Podgurski, 2003) e é apresentada uma técnica para seleção de conjuntos de dados de teste baseada em similaridade e em modelos de estados (Hemmati et al, 2010).

São descritas técnicas de geração de dados de teste: Teste Antialeatório (Malaya, 1995); e o Teste Aleatório Adaptativo (Chen et al., 2004).

Por fim, são descritos dois trabalhos, diretamente baseados em Bueno, Wong e Jino (2007), primeira publicação que descreve a técnica DOTG: o modelo de variabilidade do

teste que utiliza a medida Distância de Informação entre vetores de teste (Feldt et al., 2009); e a heurística para direcionar a busca por dados usando um conceito de estados de dados “escassos”, ou inexplorados (Alshraideh et al., 2010).

4.7.1. Seleção de Dados de Teste por Análise e Agrupamento de Perfis de Execução

Leon e Podgurski (2003) comparam empiricamente duas abordagens para filtrar e priorizar casos de teste: uma baseada na maximização da cobertura de código dos casos de teste e outra baseada em perfis de execução. Este trabalho utiliza informações sobre a execução do software para a seleção de casos de teste e aplica algoritmos de clusterização (agrupamento), mas não utiliza técnicas de otimização.

A primeira abordagem analisada seleciona casos de teste de modo que cada caso de teste cobre o maior número de comandos ainda não cobertos em relação aos casos de teste usados anteriormente. A segunda abordagem considera como os perfis de execução induzidos pelos casos de teste estão distribuídos em um espaço de perfis. Uma filtragem de clusters automaticamente particiona o conjunto de casos de teste de acordo com a similaridade dos perfis, avaliada por meio de uma medida de dissimilaridade. Testes são então selecionados de cada um dos clusters para formar o conjunto de casos de teste.

Em ambas as abordagens o objetivo é selecionar um subconjunto de teste relativamente pequeno, mas que encontre uma grande parte dos defeitos que seriam encontrados se todos os dados do conjunto de teste fossem utilizados. É assumido que existem recursos suficientes para executar e extrair o perfil de execução de todos os dados de teste do conjunto, mas que não existem recursos suficientes para avaliar manualmente todas as saídas produzidas pelas execuções.

A medida de dissimilaridade usada retorna um valor de dissimilaridade para um par de perfis de execução. A cada nó do programa é associando um contador que registra o número de execuções do nó. A dissimilaridade entre dois perfis de execução representa a

distância euclidiana em um espaço de n dimensões, sendo n o número de nós do programa. Os autores destacam que várias outras funções de dissimilaridade podem ser empregadas.

A filtragem de casos de teste baseada em perfis de execução considera a ideia de que a distribuição dos perfis de teste reflete uma informação adicional sobre os casos de teste em relação à informação de cobertura do código desses casos de teste. Busca-se assim identificar características potencialmente relevantes para revelar defeitos. Por exemplo: clusters de perfis similares podem indicar dados de teste redundantes, bastando um ou uma pequena amostra de cada cluster; perfis de execução isolados podem indicar situações muito atípicas, especialmente propensas a falhas.

Casos de teste pré-existentes para três compiladores foram utilizados no experimento de avaliação. Os resultados indicam que ambas as abordagens (a baseada na maximização de cobertura e a baseada em perfis de execução) são mais eficazes do que a seleção aleatória dos casos de teste. No entanto, as abordagens baseadas em perfis de execução tendem a ser mais eficazes quando são selecionados mais dados de teste do que os necessários para obter a máxima cobertura.

Este trabalho utiliza uma medida de dissimilaridade para selecionar dados de teste evitando redundâncias, ideia semelhante a da técnica DOTG. Contudo, a técnica de geração de dados é distinta da utilizada na técnica DOTG. Leon e Podgurski geram aleatoriamente dados de teste, executam o programa e armazenam os perfis de execução. Em um segundo passo o algoritmo de clusterização é aplicado para a seleção de dados de teste. A técnica DOTG ao contrário, gera dados de teste de alta diversidade por meio de meta-heurísticas, não sendo necessária a etapa de clusterização de dados de teste. A medida de dissimilaridade de perfis de execução poderia ser utilizada para a geração de dados de teste usando meta-heurísticas, em uma abordagem semelhante à DOTG.

4.7.2. Geração de Sequências de Estados Baseada em Similaridade

Hemmati et al (2010) propõem uma técnica para seleção de conjuntos de dados de teste baseada em similaridade e em modelos de estados. A técnica permite a geração

automática de conjuntos de teste a partir de máquinas de estados UML. Baseados no trabalho Leon e Podgurski (2003), descrito anteriormente, os autores exploram a ideia de selecionar dados de teste baseado em similaridades.

A abordagem tem como objetivo selecionar, a partir de um conjunto de teste TS que detecta um conjunto de defeitos F , um subconjunto de teste T de tamanho menor, e que encontre o maior percentual possível em relação a F . Uma medida de similaridade é aplicada para seleção dos dados de teste buscando atingir este objetivo. A medida de similaridade é calculada em relação aos caminhos executados na máquina de estados, sendo que caminhos são sequências de estados e de transições entre estados, associadas a gatilhos correspondentes (gatilhos representam condições a serem satisfeitas para que a respectiva transição ocorra).

Para cada par de caminhos tp_i e tp_j uma função $SimFunc(tp_i, tp_j)$ retorna um valor de similaridade que representa o número de gatilhos idênticos (associados a transições) entre os dois caminhos dividido pelo comprimento (numero de transições) dos caminhos. Esta função é chamada Tb (*trigger-based similarity*).

Um Algoritmo Genético é utilizado para a seleção de dados de teste. Um indivíduo do AG representa um conjunto de dados de teste T de tamanho n , que é um subconjunto do conjunto original TS . O valor objetivo para cada indivíduo consiste na soma da função de similaridade para cada par de dados de teste em P . O AG busca minimizar a função objetivo, isto é, minimizar as similaridades entre os caminhos executados no modelo de estados. Esta abordagem é chamada $TbGa$.

A abordagem $TbGa$ é avaliada empiricamente usando um componente de software de controle em C++ com 26 defeitos reais. O componente exibe comportamento complexo baseado em estados, modelado usando máquina de estados UML. Dentre outras questões abordadas no experimento, foi avaliado em que medida a seleção de dados baseada em similaridade é mais eficaz que a seleção baseada em cobertura e a seleção aleatória. Os resultados indicam que a abordagem $TbGa$ é mais eficaz do que as outras em relação à taxa de detecção de defeitos.

Embora com um foco diferente (geração de dados baseada em modelos de estados), este trabalho utiliza um conceito de similaridade entre casos de teste, semelhante ao conceito de diversidade da técnica DOTG.

4.7.3. Teste Antialeatório

No Teste Antialeatório – *Antirandom Testing* – (Malaya, 1995), são geradas sequências de teste de forma que cada novo dado de teste t_i na sequência é escolhido para maximizar a distância total entre t_i e cada um dos dados de teste executados previamente (t_0, t_1, \dots, t_{i-1}). Duas distâncias são consideradas para a geração de dados: distância cartesiana e distância Hamming. A escolha do próximo dado de teste é feita por meio de uma busca exaustiva, isto é, todos os dados de teste do domínio são investigados para a definição do próximo dado da sequência antialeatória.

Importante notar que o efeito de fazer com que cada dado de teste fique o mais longe possível de todos os dados executados previamente (ao invés de fazer com que cada dado de teste fique o mais longe possível apenas do vizinho mais próximo) tende a posicionar dados de teste em pontos extremos do domínio de entrada do programa. Este efeito é ilustrado na Figura 21 deste capítulo. Sequências antialeatórias tendem a gerar uma alocação de dados de teste mais próxima da situação 1 (dados de teste sobrepostos nos limites do domínio) do que da situação 3.

Conforme descrito neste capítulo a execução de um dado de teste coincidente com outro executado anteriormente não representa ganho de diversidade do conjunto de teste. Dados de teste repetidos também não contribuem em termos de chance de revelar de defeitos.

4.7.4. Teste Aleatório Adaptativo

O Teste Aleatório Adaptativo (*Adaptive Random Testing* – ART) modifica o teste aleatório por meio de um procedimento para selecionar um novo dado de teste a ser

adicionado ao conjunto de teste. A técnica define dois conjuntos de teste: o “conjunto de testes executados (CTE)” que contém todos os dados de teste executados durante o teste, e o “conjunto de testes candidatos (CTC)”, que contém um conjunto de dados de teste gerados aleatoriamente. O CTE é incrementalmente atualizado com elementos selecionados do CTC. Em cada incremento um novo dado de teste selecionado para ser inserido no CTE. Este dado de teste selecionado é aquele que apresenta o maior valor de distância mínima considerando as distâncias do dado de teste de CTC para todos os dados de teste do conjunto CTE. Em Chen et al (2004) a distância euclidiana entre dados de teste é utilizada para calcular essas distâncias. Esta abordagem é chamada especificamente de “ART – Conjunto Candidato de Tamanho Fixo” (*FSCS-ART: Fixed Size Candidate Set ART*).

Deste modo, a técnica ART busca maximizar a distância mínima do próximo dado de teste em relação aos dados de teste já executados. A lógica desta técnica é que selecionar dados de teste que não estejam muito próximos de outros dados já gerados tende a melhorar a eficácia do teste. Esta lógica está embasada essencialmente na intuição de que a existência de regiões de falha faz com que um novo dado “próximo” dos anteriores é pior do que outro dado “longe” destes. Com esta técnica Chen et al (2004) buscam uma maior eficácia do que o Teste Aleatório. Como métrica de eficácia é proposta a *medida-F*, que representa o número de dados de teste necessários para detectar a primeira falha. Um menor valor para a medida-F significa um número menor de dados de teste necessários para provocar a primeira falha.

Variações da técnica FSCS-ART foram propostas:

Chen et al (2003) buscam reduzir o custo de geração de dados desta técnica com a ART-Espelho (*MART - Mirror Adaptive Random Testing*). Nesta abordagem o domínio de entrada do programa é dividido em m subdomínios disjuntos. Um subdomínio é designado como original e o algoritmo ART original (FSCS-ART) é aplicado apenas neste subdomínio original. Cada dado de teste gerado no subdomínio original é espelhado sucessivamente nos outros subdomínios (chamados subdomínios espelhos). Funções de espelhamento são usadas para, por exemplo, transportar linearmente um dado de teste de um subdomínio para outro.

Chen et al (2003) propõem a técnica ART por particionamento aleatório (ART by Random Partitioning). Nesta abordagem o dado de teste utilizado mais recentemente é usado para subdividir o domínio de entrada. Um novo dado de teste é então selecionado aleatoriamente do maior entre os subdomínios remanescentes. Uma variação desta ideia é apresentada em Chen (2004), chamada de ART por localização (ART by Localization), que envolve dois aspectos: 1) restringir a seleção de dados de teste à parte do domínio de entrada onde é mais provável gerar algum dado de teste distante dos dados já executados; e 2) restringir os cálculos de distância aos dados de teste que estão próximos à região da geração de dados.

Mayer (2005) apresenta o Teste Aleatório Adaptativo Baseado em *Lattices* (*Lattice-Based ART*). O algoritmo de geração de dados realiza várias varreduras no domínio de entrada, em cada varredura são gerados dados em uma estrutura regular do tipo Lattice (ou grid). A cada varredura o lattice é progressivamente refinado com dados de teste gerados em uma ordem aleatória. Cada dado de teste é submetido a uma pequena perturbação aleatória visando introduzir aleatoriedade nos dados de teste gerados.

Chen e Merkel (2007) apresentam uma variação da técnica ART baseada em sequências de baixa dispersão chamadas de quasi-aleatórias. Os autores caracterizam estas sequências como “uma sequência de pontos em um cubo b-dimensional que têm a propriedade de, em qualquer subintervalo, os pontos serem razoavelmente uniformemente distribuídos”. É utilizado um gerador de sequências quasi-aleatórias disponível na biblioteca científica GNU, implementada em C. Uma limitação da técnica é que o gerador utilizado gera apenas uma única sequência, independente da semente utilizada, o que impede a avaliação empírica da técnica com vários conjuntos de teste. Para tratar esta limitação são aplicados métodos de introdução de perturbações aleatórias (referidas como *Scrambling*) para gerar sequências de baixa discrepância *aleatorizadas*.

Trabalhos relacionados à técnica ART tipicamente são avaliados por meio de simulações, ou com a aplicação da técnica em pequenos programas de tratamento numérico. Por exemplo, a avaliação empírica da abordagem ART – Conjunto Candidato de Tamanho Fixo (*FSCS-ART*) compara a eficácia desta técnica em relação ao teste aleatório

(*RT*) com o uso da medida-F de eficácia. Foram utilizados 12 pequenos programas numéricos com alguns defeitos inseridos (entre três e nove defeitos por programa). O tamanho dos programas varia de 30 a 200 comandos, e o domínio de entrada possui dimensão entre um e quatro.

Foi utilizado um conjunto candidato com 10 dados de teste ($|CTC| = 10$). Para cada programa foram aplicados conjuntos de teste *FSCS-ART* e conjuntos *RT* e foram obtidos pares (u^a , u^r), em que u^a refere-se ao número de dados de teste necessários para detectar a primeira falha usando conjuntos *FSCS-ART* e u^r refere-se ao número de dados de teste necessários para detectar a primeira falha usando conjuntos *RT*. Esses valores (u^a , u^r) foram obtidos em 3000 repetições, com a semente aleatória variando em cada repetição e valores médios foram computados gerando valores chamados *Fa* e *Fr*. Segundo os autores na maioria das situações os conjuntos *FSCS-ART* foram mais eficazes com um fator que varia de um programa para outro, de 1% a valores da ordem de 40%.

Teste Aleatório Adaptativo e Diversidade

As técnicas *ART* e a técnicas *DOTG* têm em comum a utilização de ideias do teste aleatório (*RT*) e da tentativa de aprimorá-lo. Estas duas técnicas são fundamentalmente diferentes em alguns pontos.

Na técnica *DOTG* (que gera conjuntos de teste *DOTS*) os conjuntos de teste gerados aleatoriamente são incrementalmente aprimorados pelo reposicionamento dos dados de teste do conjunto de teste – realizado a cada nova iteração das meta-heurísticas de geração de dados – de uma maneira tal que aumenta progressivamente os valores de diversidade desses conjuntos de teste. Na observação cuidadosa do funcionamento das meta-heurísticas fica claro que estes movimentos de progressivo refinamento são essenciais para que se atinja uma alta diversidade “global” dos dados de teste e evitar partes do domínio de entrada com baixa diversidade (“empilhamentos” de dados de teste em pequenas regiões). As técnicas *ART*, independentemente da variação considerada, não permitem este reposicionamento: uma “escolha ruim” para uma localização de dados de teste não pode ser

corrigida em um passo posterior. Além deste aspecto, as técnicas ART geram um dado de teste usando informação sobre as posições dos dados de teste previamente gerados, ao contrário, a técnica DOTG, permite levar em conta todos os dados de teste do conjunto para otimizar a diversidade.

A abordagem orientada à diversidade pode ser definida com o uso de vários tipos de informação, o que resulta em diferentes perspectivas para a diversidade. Por outro lado as técnicas ART (e também a Antialeatória) apresentam apenas uma perspectiva (de como os pontos de teste cobrem o domínio de entrada). Portanto, a técnica DOTG pode ser vista como uma “abordagem geral de diversidade”, enquanto que as técnicas ART, Antialeatória, e DOTG-ID representam “abordagens específicas de diversidade” que focam no domínio de entrada do programa.

As técnicas ART e Antialeatória podem ser utilizadas para gerar dados de teste, mas não para avaliar um conjunto de teste, gerado com a utilização de outra técnica. Ao contrário, a abordagem DOTG além de permitir a geração de conjuntos de teste de alta diversidade, pode ser utilizada também para avaliar a diversidade de um conjunto de teste, gerado por meio de qualquer outra técnica de geração de dados. Por exemplo, se dois conjuntos de teste – T1 e T2 – satisfazem o critério Todos os Ramos para um programa, os valores de diversidade dos conjuntos de teste podem ser computados – $Div(T1)$, $Div(T2)$ – e ser utilizados como um critério ortogonal para a escolha de qual conjunto de teste será utilizado. Esta característica amplia o leque de situações em que a DOTG pode ser usada.

A partir da diversidade do conjunto de teste e do tamanho deste conjunto, é possível calcular a lacuna média existente entre dados de teste adjacentes no domínio de entrada multidimensional (efeito discutido na Seção 4.6). Esta informação pode ser utilizada para prever o tamanho das regiões de falha a serem encontrados com a aplicação do conjunto de teste. Este efeito foi observado nos experimentos realizados (próximo capítulo), no entanto, a utilização desta característica da técnica DOTG na prática requer mais análises. Cabe destacar que a técnica DOTG é única ao estabelecer relação entre uma propriedade mensurável de um conjunto de teste (a diversidade do conjunto) e o tamanho provável de regiões de falha que serão reveladas no teste.

4.7.5. Medida de Variabilidade do Teste com o uso de Distância de Informação

Feldt et al (2009) propõem um modelo de variabilidade do teste utilizado para enfatizar a seleção de dados de teste que são diferentes em relação aos dados já executados. A métrica proposta calcula a distância entre dois vetores de teste usando a Distância de Informação, uma métrica baseada na complexidade de Kolmogorov. Como esta complexidade é não computável os autores propõem o uso de compressores de dados para obter uma aproximação dos valores de Distância de Informação entre dados de teste. É afirmado que tal métrica captura “diferenças cognitivas” entre os dados de teste baseada em informações dos traços de execução desses dados. Os traços de execução podem representar diferentes informações como de fluxo de controle, ou de mudanças nos estados dos dados. Não está claro no trabalho como os traços de execução são organizados e como as distâncias são calculadas.

A abordagem foi aplicada para calcular distâncias entre 25 dados de teste para o programa de classificação de triângulos. Os resultados de como os dados de teste são agrupados e classificados com o uso da métrica são comparados com os resultados de como três humanos avaliam e agrupam os mesmos dados de teste. A conclusão é que a aplicação da métrica resulta na identificação de similaridades de dados de teste semelhante à identificação de similaridades realizada pelos humanos.

Neste trabalho não é feita a proposta de algoritmos para gerar dados de teste utilizando a métrica proposta. Também não são realizadas avaliações sobre a eficácia dos dados de teste gerados. Como os autores reconhecem, não é claro se as diferenças entre dados de teste do ponto de vista da teoria da informação são relevantes para a qualidade do teste, e se testes que aparentem serem diversos para humanos sejam mais eficazes no teste.

4.7.6. Utilização de Estados de Dados Escassos para Direcionar a Busca de Dados de Teste

Alshraideh et al (2010) apresentam uma heurística para direcionar a busca por dados de teste quando a função objetivo (ou função *fitness*) associada a um objetivo de teste é incapaz de distinguir a qualidade das soluções candidatas (dados de teste). Este problema pode ocorrer em abordagens de Geração de Dados Baseada em Busca que usam uma função objetivo para avaliar a qualidade de dados de teste para exercitar um objetivo definido, por exemplo, exercitar um ramo de um programa. Se o predicado do ramo leva a uma função objetivo que retorna valores constantes, é difícil discriminar a qualidade entre as soluções candidatas e direcionar a busca. A heurística proposta é aplicada quando esta situação é identificada (função objetivo com valores constantes) e produz estados de dados “escassos”, ou inexplorados. A lógica é que estados de dados que ocorrem comumente já teriam sido encontrados e utilizados durante a busca, enquanto que estados de dados escassos são mais prováveis de levar a melhorias na função objetivo.

A busca por estados de dados escassos (estados raros) é iniciada quando a função objetivo associada a um ramo torna-se constante durante a busca. Os valores das variáveis (os estados de dados) produzidos durante as execuções são registrados, agrupados em classes, e representados como conjuntos de histogramas. Estes histogramas são utilizados para computar medidas de distância e enfatizar a seleção de estados de dados escassos. A abordagem de Alshraideh et al. foi avaliada empiricamente e os resultados sugerem que há busca de estados de dados escassos é eficaz para discriminar dados de teste e exercitar ramos de programas.

Uma clara diferença entre abordagem proposta da geração de dados orientada à diversidade (DOTG) e a heurística de Alshraideh et al é que a identificação de estados de dados escassos busca “complementar” a busca com valores pouco explorados, visando justamente a promover a diversidade de estados de dados. Esta análise dos estados de dados é feita em estados intermediários de execução, enquanto que na abordagem DOTG-ID é feita nos dados de entrada do programa. Por fim, os objetivos são também distintos.

Enquanto a DOTG almeja aumentar a eficácia do teste em termos de habilidade de revelar defeitos, a heurística de Alshraideh et al. visa a dar direcionamento na busca de valores específicos (escassos) quando a função objetivo retorna valores constantes.

4.8. Considerações Finais

Este capítulo apresentou a Geração de Dados de Teste Orientada à Diversidade (DOTG). A intuição relacionada a esta técnica é de que a variabilidade (diversidade) dos dados de teste utilizados para testar um software tem um papel relevante na qualidade do teste. Em geral, não há informação exata sobre onde estão os defeitos do software. Nesta situação de incerteza, diversificar o teste pode ser uma estratégia mais prudente.

A intuição inicial descrita acima ganhou apoio ao se considerar trabalhos em diferentes áreas: confiabilidade de software, tolerância a defeitos e teste baseado em particionamento, segundo descrito na Seção 4.2.2 e detalhando no Apêndice C.

Várias perspectivas da abordagem DOTG podem ser desenvolvidas por meio da utilização de informações de diferentes naturezas para calcular a diversidade de conjuntos de teste. O “meta-modelo” para a diversidade fornece um guia para o desenvolvimento de perspectivas da abordagem DOTG.

A perspectiva do domínio de entrada para a Geração de Dados de Teste Orientada à Diversidade (DOTG-ID) foi desenvolvida. Esta perspectiva considera o posicionamento dos dados de teste no domínio de entrada do programa para calcular a diversidade de um conjunto de teste. Esta perspectiva (DOTG-ID) foi escolhida para o desenvolvimento devido aos seguintes fatores:

- Trata-se de uma perspectiva “mais simples”, visto que não é necessário executar código ou modelos para avaliar a diversidade. Este é um ponto importante: é possível gerar conjuntos DOTS mesmo não tendo o código fonte, e sem a necessidade de desenvolver modelos. A rigor, a DOTG-ID requer como informação apenas a descrição das variáveis de entrada recebidas pelo software.

- O conceito de região de falhas – em especial a continuidade dessas regiões – é avaliado no domínio de entrada do programa (regiões de falha estão no domínio de entrada). Pareceu natural, portanto, utilizar a base conceitual fornecida por esses trabalhos para definir a DOTG-ID. A avaliação empírica realizada (próximo capítulo) sugere que as características dos conjuntos DOTS na abordagem DOTG-ID aumentam a eficácia para encontrar regiões de falha ¹⁰.

Foram desenvolvidas três meta-heurísticas para a geração automática de conjuntos de teste de alta diversidade (DOTS): a SA-DOTG, baseada em Recozimento Simulado; a GA-DOTG, baseada em Algoritmo Genético; e a SR-DOTG, baseada na dinâmica de sistemas de partículas.

As meta-heurísticas SA-DOTG e GA-DOTG podem ser vistas como ferramentas genéricas. São potencialmente aplicáveis para a geração de conjuntos DOTS para qualquer perspectiva (também para a perspectiva do domínio de entrada – DOTG-ID). Ambas abordagens utilizam o valor de diversidade dos conjuntos de teste para direcionar a busca para a obtenção dos conjuntos DOTS a partir de conjuntos de teste gerados aleatoriamente (RTS).

A meta-heurística Repulsão Simulada para geração de dados de teste (SR-DOTG) é específica para perspectiva do domínio de entrada – DOTG-ID. A SR-DOTG também parte de conjuntos RTS, mas não há necessidade de direcionar a busca com o uso dos valores de diversidade. A simulação de repulsão entre os dados de teste faz com que eles se posicionem “espontaneamente” no domínio de entrada do programa, de forma a otimizar a diversidade do conjunto de teste.

Um ponto de atenção relacionado à técnica DOTG-ID é que exercitar de forma rigorosa e completa um domínio de entrada tende a ter um alto custo. O número de dados de teste necessários para que se ganhe expressiva confiança no software pode ser muito

¹⁰ (Nota: Cabe observar que modelos de detecção de defeitos podem ser estendidos buscando caracterizar “regiões de erro”, isto é, regiões associadas a estados internos de execução incorretos, que podem levar a uma falha. Modelos de “regiões de erro” poderiam ser utilizados para definir outras perspectivas de diversidade. Por exemplo, para exercitar de forma diversa os estados internos de execução do software – e não apenas estados diversos no domínio de entrada – Vide Apêndice D).

grande, sempre há possibilidade de regiões de falha muito pequenas não serem detectadas com os conjuntos DOTS. Para ilustrar esta questão, basta considerar que cada dimensão do domínio de entrada tem o impacto de dobrar o número de pontos extremos do domínio. Ou seja, o tamanho deste domínio cresce exponencialmente com o número de variáveis de entrada. Uma discussão sobre a complexidade das meta-heurísticas e sobre o custo computacional para a geração de conjuntos DOTS é fornecida no próximo capítulo.

Mesmo que o custo da geração dos dados seja relativamente pouco expressivo (tarefa realizada mecanicamente pelas meta-heurísticas), a execução e a avaliação da saída produzida em grandes volumes de dados tende a ser muito onerosa.

Este fato, contudo, aplica-se também ao Teste Aleatório, e às técnicas baseadas em particionamento (ex; teste baseado em fluxo de dados e análise de mutantes). Trata-se em última análise, do reconhecimento da impossibilidade do teste exaustivo, limitação destacada por Dijkstra (1972): “o teste pode apenas mostrar a presença de um erro; o teste nunca pode provar a ausência de todos os erros”.

O capítulo seguinte apresenta avaliações empíricas realizadas para investigar o desempenho das meta-heurísticas e para avaliar a abordagem DOTG-ID.

Capítulo 5 – Avaliação Empírica da Geração de Dados de Teste Orientada à Diversidade

"It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts".

Arthur Conan Doyle via Sherlock Holmes.

Este capítulo apresenta as avaliações empíricas realizadas para analisar características da técnica de teste proposta, a Geração de Dados de Teste Orientada à Diversidade (*Diversity Oriented Test Data Generation – DOTG*) e das meta-heurísticas implementadas para a geração de conjuntos de teste de alta diversidade (DOTS). Para uma breve discussão sobre avaliações empíricas em engenharia de software, favor rever o Capítulo 2.

Na Seção 5.1 apresenta-se a avaliação do desempenho das meta-heurísticas implementadas para gerar conjuntos de teste de alta diversidade – DOTS; a Seção 5.3 descreve uma simulação Monte Carlo realizada com o objetivo de estudar a influência dos fatores: tipo de região de falha; taxa de falhas; e tamanho do conjunto de teste na eficácia

das técnicas de teste (Metropolis, e Ulam, 1949). A Seção 5.5 descreve um experimento realizado para avaliar o efeito da diversidade dos conjuntos de teste na cobertura alcançada no teste de programas, medida considerando critérios de teste baseados em análise de fluxos de dados e no critério baseado em defeitos Análise de Mutantes.

5.1. Desempenho das Meta-heurísticas para Gerar Diversidade

Um primeiro aspecto avaliado empiricamente foi a eficiência das meta-heurísticas para gerar conjuntos de teste de alta diversidade (os conjuntos DOTS-ID, ou simplesmente DOTS). As três meta-heurísticas implementadas (SA-DOTG, GA-DOTG, e SR-DOTG) foram utilizadas para geração de dados de teste em uma série de tentativas com o objetivo de ajustar parâmetros de busca específicos. Nesta fase o objetivo foi o de definir valores para os parâmetros que permitissem um bom desempenho para otimizar valores de diversidade dos conjuntos de teste.

Após esta fase inicial, utilizando os valores definidos para parâmetros de cada meta-heurística, foram feitas várias tentativas de geração de dados de teste a fim de avaliar a influência de aspectos do problema no desempenho das meta-heurísticas. Foram avaliados os aspectos: tipo das variáveis de entrada dos dados de teste (inteiros, reais e combinação de tipos); número de variáveis de entrada em cada dado de teste (este valor caracteriza a dimensão do domínio de entrada do programa – o domínio de busca); e o tamanho dos conjuntos de teste.

Tabela 2. Número de iterações e valores de diversidade

Nº da Iteração	SA-DOTG	GA-DOTG	SR-DOTG
50	5585.3	6651.6	10514.1
100	5741.1	6946.8	10594.2
500	6523.7	7883.3	10651.4
1000	6523.7	8174.2	10662.3
2000	6526.1	8685.8	10673.5
4000	7526.1	8973.8	10673.5

Para avaliar quantitativamente a eficiência das meta-heurísticas, gerações de dados de teste foram realizadas e foram registrados os valores de diversidade alcançados pelas meta-

heurísticas. A Tabela 2 exemplifica os resultados alcançados. A Tabela mostra valores médios de diversidade alcançados para cada meta-heurística (Equação 4.2, Capítulo 4) em números de iterações específicos (coluna da esquerda). Cada célula da tabela refere-se a uma média de 30 execuções. Em todos os casos tem-se um domínio de entrada bidimensional com duas variáveis do tipo real variando de 0 a 1024; com 100 dados de teste ($NDT = 100$). A diversidade de conjuntos gerados aleatoriamente (RTS) é da ordem de 5100.

A análise da tabela permite afirmar que na implementação atual das meta-heurísticas a SR-DOTG é superior à GA-DOTG, que por sua vez é superior à SA-DOTG. Com apenas 50 iterações a SR-DOTG alcança valores de diversidade superiores aos obtidos pela GA-DOTG e SA-DOTG em 4000 iterações. Embora os dados se refiram a uma situação específica, a variação dos aspectos considerados (tipo de variável, número de variáveis de entrada, número de dados de teste, etc.) não interfere na ordem relativa de eficiência das meta-heurísticas.

A complexidade computacional da SR-DOTG é maior do que as complexidades da GA-DOTG e da SA-DOTG. O custo do cálculo de forças e movimentos da SR-DOTG cresce com o quadrado do número de dados de teste no conjunto (NDT). Este custo cresce também linearmente com o número de conjuntos de teste (NTS) e com o número de variáveis de entrada nos dados de teste (NIV). O custo computacional das meta-heurísticas GA-DOTG e SA-DOTG, ao contrário da SR-DOTG, cresce apenas linearmente com esses parâmetros (NDT , NIV , e também NTS no caso do GA-DOTG).

Apesar do maior custo computacional da SR-DOTG, o fato desta meta-heurística ser amplamente mais eficiente que as outras, faz com que a geração de dados com a SR-DOTG seja mais rápida do que com as GA-DOTG e SA-DOTG, além de serem alcançados valores significativamente maiores de diversidade. Por exemplo, para gerar o conjunto de teste mostrado na Figura 23, com 100 dados de teste, a SR-DOTG requer aproximadamente 200 iterações, 0.4 segundos, para otimizar a diversidade do conjunto de teste de valores da

ordem de 522 (conjuntos de dados gerados aleatoriamente – RTS) para valores da ordem de 1063 (conjuntos de dados de alta diversidade DOTS) ¹¹.

Como resultado desta análise, decidiu-se utilizar apenas a SR-DOTG como ferramenta para as análises empíricas adicionais, descritas a seguir, referentes à perspectiva do domínio de entrada do programa para diversidade (DOTG-ID). Os resultados obtidos com SR-DOTG poderiam ser obtidos também com as outras meta-heurísticas, visto que as características dos conjuntos de alta diversidade são semelhantes. Neste caso seria necessário um número de iterações significativamente maior, assim como o tempo de processamento.

Importante notar que esta maior eficiência da SR-DOTG não torna desnecessárias as meta-heurísticas GA-DOTG e SA-DOTG. Conforme destacado no Capítulo 4, a SR-DOTG é específica para a perspectiva em consideração para a análise empírica (a DOTG-ID), ao passo que GA-DOTG e SA-DOTG são genéricas e potencialmente aplicáveis a qualquer perspectiva de diversidade e qualquer função a ser otimizada.

5.2. Simulação Monte Carlo para Avaliar Fatores que Influenciam a Eficácia da DOTG-ID

No Capítulo 4, Seção 4.5, foram descritos trabalhos que avaliam propriedades de regiões de falha presentes nos domínios de entrada do software. A noção de região de falha é definida por Frankl et al. como “o conjunto de pontos de falha que é eliminado por uma correção do programa” (Frankl et al., 1997). Ammann e Knight (1988) definem o mesmo conceito como “um domínio de falhas com a sua geometria”, sendo que domínio de falhas é o conjunto de pontos de entrada que causam a falha do programa. Importante notar também que regiões de falha caracterizam o subconjunto do domínio de entrada do programa que satisfaz as condições para a revelação de defeitos, descritas em modelos como o de Clarke: executar o comando defeituoso, criar um estado incorreto de execução, e propagar este

¹¹ PC Itautec, processador Intel Core 2 Duo CPU E4500, 2.20 Ghz, 1.99 GB de RAM.

estado para alguma saída observável (Thompson et al., 1993) – veja Capítulo 4, Seção 4.5.2.

Com o propósito de avaliar técnicas de teste é possível realizar uma simulação de Monte Carlo pela geração aleatória de “regiões de falha” em um “domínio de entrada do programa” (Metropolis e Ulam, 1949), (Metropolis et al., 1953), (Caflisch, 1998), (Press et al., 1992).

Regiões de falha definem um subconjunto do domínio de entrada no qual todos os dados de teste provocam a falha do programa. Depois de criada uma região de falha, dados de teste podem ser gerados e avaliados. Um dado de teste “provoca a falha” se ele está contido na região de falha, e não provoca a falha caso contrário. (Notar que como se trata de simulação não há defeitos de fato, nem regiões de falha de fato, mas sim regiões de falha simuladas. Além disto, trata-se de um domínio de entrada fictício, sem um programa associado).

O objetivo da avaliação realizada é estudar a influência dos fatores: tipo de região de falha; taxa de falhas; e tamanho do conjunto de teste na eficácia da técnica DOTG-ID. A título de comparação foi considerada a eficácia esperada de conjuntos de teste gerados aleatoriamente (RTS).

Esta abordagem baseada em simulação para avaliar conjuntos de teste foi aplicada, por exemplo, em: (Chen et al., 2005), (Mayer, 2005), (Chen e Kuo, 2006), (Chen et al., 2006), (Mayer, 2006) e (Chen e Merkel, 2007). Muller e Pfahl (2008) destacam a importância de simulações para apoiar estudos em engenharia de software, em especial para analisar o comportamento dinâmico de processos complexos, permitido refinar premissas e apoiar análises, estimativas, e decisões, relativas ao objeto do estudo.

A avaliação descrita nesta seção utiliza os mesmos padrões de região de falha propostos por estes autores, já descritos no Capítulo 4, Seção 4.5.3: falhas tipo Ponto, tipo Faixa e tipo Bloco. Relembrando, no padrão tipo *Ponto* as entradas causadoras de falha são pontos, regiões de tamanho muito pequeno espalhadas no domínio de entrada. No padrão tipo *Faixa* a região de falha tem a forma de uma faixa fina. O padrão do tipo *Bloco*

concentra entradas causadoras de falhas na forma de blocos. O capítulo citado apresenta ilustrações desses tipos de regiões e programas que contém defeitos que as provocam.

Na simulação realizada foi utilizado um domínio de entrada de duas dimensões relacionadas a duas variáveis do tipo real. Ambas as variáveis variam de 0.0 a 100.0. As taxas de falha (θ) são dadas pela razão entre a área da região de falha e a área de todo o domínio. Por exemplo, neste domínio de entrada, se a região de falha é do tipo bloco e com medidas 10.0 por 10.0, a taxa de falhas será 0.01 ($\theta = 10^2/100^2$).

Falhas padrão tipo bloco são geradas pela seleção aleatória de um ponto âncora no domínio de entrada e definição de um quadrado centralizado no ponto âncora. A dimensão do lado do quadrado é determinada de forma a resultar em uma área específica, relacionada à taxa de falhas desejada (θ). Se alguma parte do bloco estiver localizada fora do domínio de entrada, um ajuste no formato do bloco (alongamento em uma das dimensões) é feito, mantendo a área total inalterada e localizada dentro do domínio (conforme descrito acima, a área do bloco define a taxa de falhas). Para falhas padrão tipo faixa dois pontos âncora são selecionados aleatoriamente em dois lados adjacentes do domínio e uma faixa estreita conecta esses pontos. A largura desta faixa é calculada de forma a resultar em uma área específica, relacionada à taxa de falhas desejada. Pontos muito próximos dos cantos do domínio são evitados, pois eles resultariam em partes da faixa fora do domínio de entrada. Falhas tipo ponto são criadas em dez pontos âncora selecionados aleatoriamente no domínio. Os raios de todos os círculos são iguais e calculados de forma a resultar na taxa de falhas desejada. Caso algum círculo for sobrepor algum outro, ou se algum círculo for exceder os limites do domínio, um novo ponto âncora é aleatoriamente gerado em substituição ao anterior.

Valores utilizados para as taxa de falhas (θ) são: 0.002; 0.01 e 0.05. Valores deste nível de grandeza para θ têm base realística e são utilizados em outros trabalhos, ex: (Ntafos, 1998). Para cada valor de taxa de falhas foram utilizados vários tamanhos de conjuntos de teste (|T|). |T| assume valores entre 20 e 240 com incrementos de 20. A ordem de magnitude de NTD foi definida visando produzir uma “chance média” de provocar falhas. Esta escolha visa a permitir que possíveis diferenças entre as técnicas de teste

(DOTG-ID e RT) se tornem perceptíveis (valores extremos para $|T|$ tendem a reduzir as diferenças entre as técnicas, vide análise nas seções seguintes).

O procedimento de avaliação é o seguinte. Para cada taxa de falhas e tamanho do conjunto de teste foram realizadas 2000 tentativas (execuções da meta-heurística SR-DOTG). Em cada tentativa são gerados NTS ($NTS = 10$) conjuntos de teste aleatórios (chamados RTS) utilizando um gerador aleatório uniforme com reposição (cada valor de cada variável tem a mesma chance de ser selecionado, o mesmo valor pode ser selecionado mais de uma vez). Cada conjunto de teste gerado aleatoriamente RTS_i ($i = 1 \dots NTS$) tem a sua diversidade otimizada pelo SR-DOTG (a otimização ocorre concorrentemente e de forma independente para todos os conjuntos de teste), ao final da otimização o conjunto de mais alta diversidade dentre os NTS conjuntos de alta diversidade – $DOTS_i$ ($i = 1 \dots NTS$) é selecionado para avaliação.

Os conjuntos DOTSS selecionados são avaliados para os três tipos de padrão de falha. Para cada conjunto de teste três diferentes regiões de falha são geradas, cada região de um padrão distinto (padrões tipos: bloco, faixa, ponto), conforme descrito anteriormente. Um contador é associado a cada tipo de padrão de falha e é incrementado se algum dado de teste do conjunto está localizado dentro da região de falha (basta um dado de teste na região de falha para que o contador seja incrementado, além disto, o incremento será sempre de 1, mesmo se mais de um dado de teste estiverem na região de falha). Após todas as tentativas os valores dos contadores são utilizados para calcular a medida de eficácia dos DOTSS com respeito a cada padrão de falha distinto e tamanho do conjunto de teste. Os parâmetros específicos de busca do SR-DOTG são os seguintes: carga inicial variando de 1000 a 10000; número limite de iterações 600 (em cada tentativa o SR-DOTG executa 600 iterações para, partindo dos RTS, obter os DOTSS); e coeficiente de descarga entre 0.95 e 0.99.

Foi definida como medida de eficácia a proporção de conjuntos de teste (DOTSS) que encontram a região de falha (“revelam a falha”). Valores de eficácia são calculados para cada tamanho de conjunto de teste e para cada taxa de falhas. Esses valores podem variar de zero, caso nenhum conjunto de teste DOTSS encontre falha, até um, caso todos os DOTSS

encontrem a falha (são 2000 tentativas, portanto eficácia igual a 1.0 significa que todos os 2000 conjuntos de teste encontraram a falha).

Foi calculada também a razão de eficácia entre os conjuntos DOTS e conjuntos gerados aleatoriamente (RTS) – isto é, a razão entre o número de conjuntos DOTS que encontram a falha e o número de conjuntos RTS que encontram falhas. A eficácia esperada de conjuntos de teste gerados aleatoriamente é calculada usando a seguinte fórmula:

$$ER(\theta, T) = (1 - (1 - \theta)^{|T|}) 2000 \quad (5.1)$$

ER é o número esperado de conjuntos de teste RTS que revelam falhas, sendo $|T|$ o número de dados de teste do conjunto e θ a taxa de falhas, ex: (Weyuker e Jeng, 1991), (Ntafos, 1998), (Ntafos, 2001). Notar que, para conjuntos de dados de teste gerados aleatoriamente (RTS), o tipo de padrão de falha (tipo Ponto, tipo Faixa, ou tipo Bloco) não interfere na eficácia dos conjuntos de teste. Para esses conjuntos a eficácia (valor ER) depende apenas da taxa de falhas associada à região de falha e do tamanho do conjunto de teste.

5.3. Resultados da Simulação

As Figuras 31(a), 31(b) e 31(c) mostram a eficácia dos conjuntos DOTS e a eficácia esperada do teste aleatório (conjuntos RTS) para cada tipo de padrão de falha e para vários tamanhos de conjuntos de teste (eficácia dos conjuntos RTS calculada por meio da Equação 5.1). Os resultados em 31(a), 31(b) e 31(c) referem-se a taxas de falhas (θ) iguais a 0.002, 0.01 e 0.05, respectivamente. A razão de eficácia entre os conjuntos DOTS e os conjuntos RTS são mostradas nas figuras 32(a), 32(b) e 32(c), (próximas páginas) para os mesmos padrões de falha, tamanhos de conjuntos de teste e taxas de falhas.

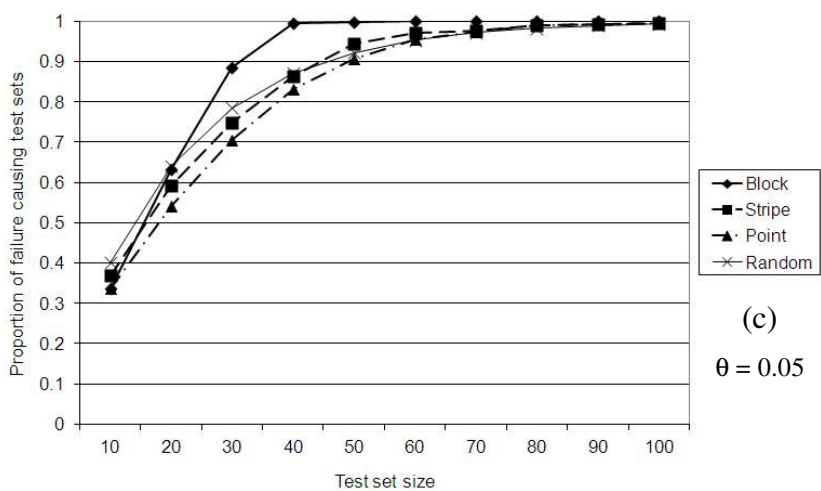
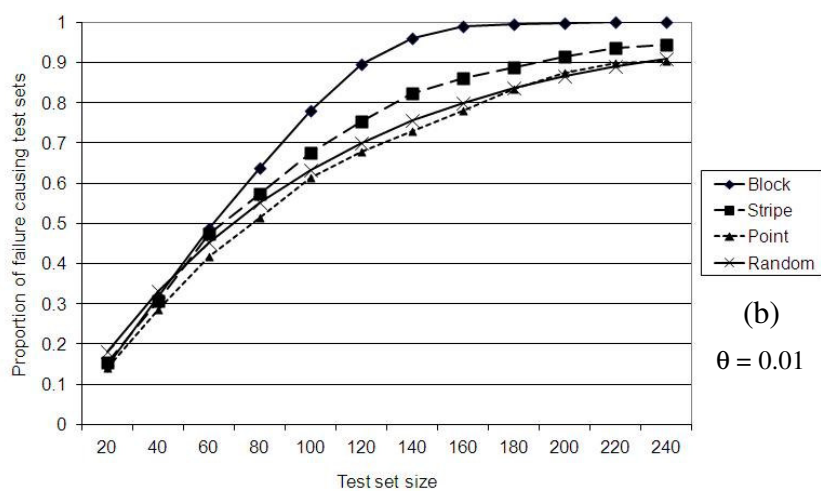
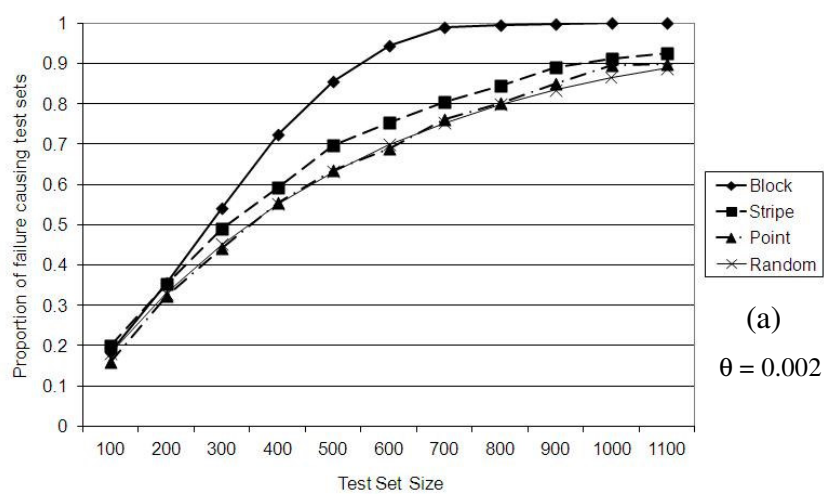
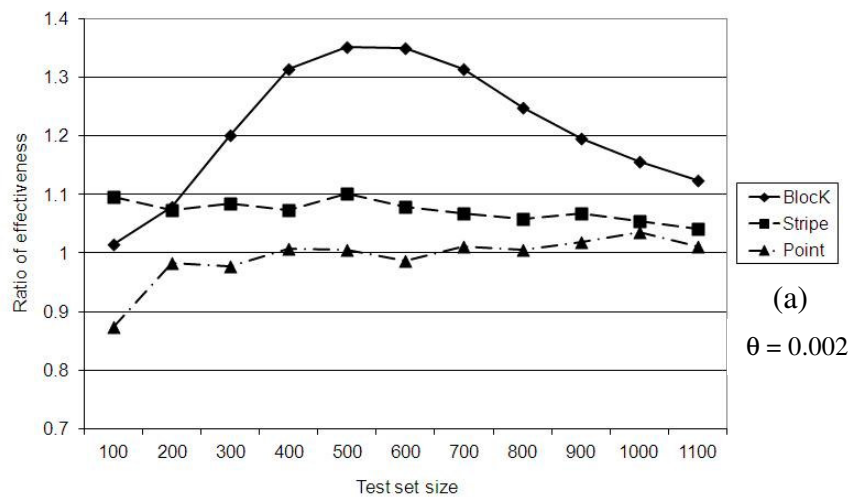


Figura 31. Eficácia de conjuntos DOTS e RTS

As Figuras 31(a), 31(b) e 31(c) mostram que conjuntos de teste DOTS para o padrão de falhas do tipo bloco apresentam uma eficácia significativamente superior a dos conjuntos de teste RTS. Esta superioridade é maior para falhas menores (com valores menores de taxas de falhas $\theta = 0.002$ e $\theta = 0.01$) e para tamanhos médios de conjuntos de teste. Para padrões do tipo faixa os resultados são similares, mas a superioridade em eficácia dos conjuntos DOTS em relação aos conjuntos RTS é menor, para todas as taxas de falhas e tamanhos de conjunto de teste. Para o padrão de falhas tipo ponto os conjuntos RTS mostram-se ligeiramente superiores aos conjuntos DOTS, esta superioridade deixa de existir para taxas de falhas menores ($\theta = 0.002$), situação na qual as duas técnicas mostram-se equivalentes.

A utilização de valores extremos para $|T|$ para um θ fixo, tende a fazer com que a eficácia dos conjuntos DOTS e RTS se aproxime. Intuitivamente isto ocorre por que com $|T|$ baixo ambas as técnicas têm chance muito pequena de revelar defeitos. Ao contrário, com $|T|$ tendendo para valores muito grandes, a partir de um certo valor de $|T|$ os conjuntos DOTS “saturam” atingem máxima eficácia, enquanto que os RTS, menos eficazes, ainda permitem aumento da eficácia com o aumento de $|T|$.



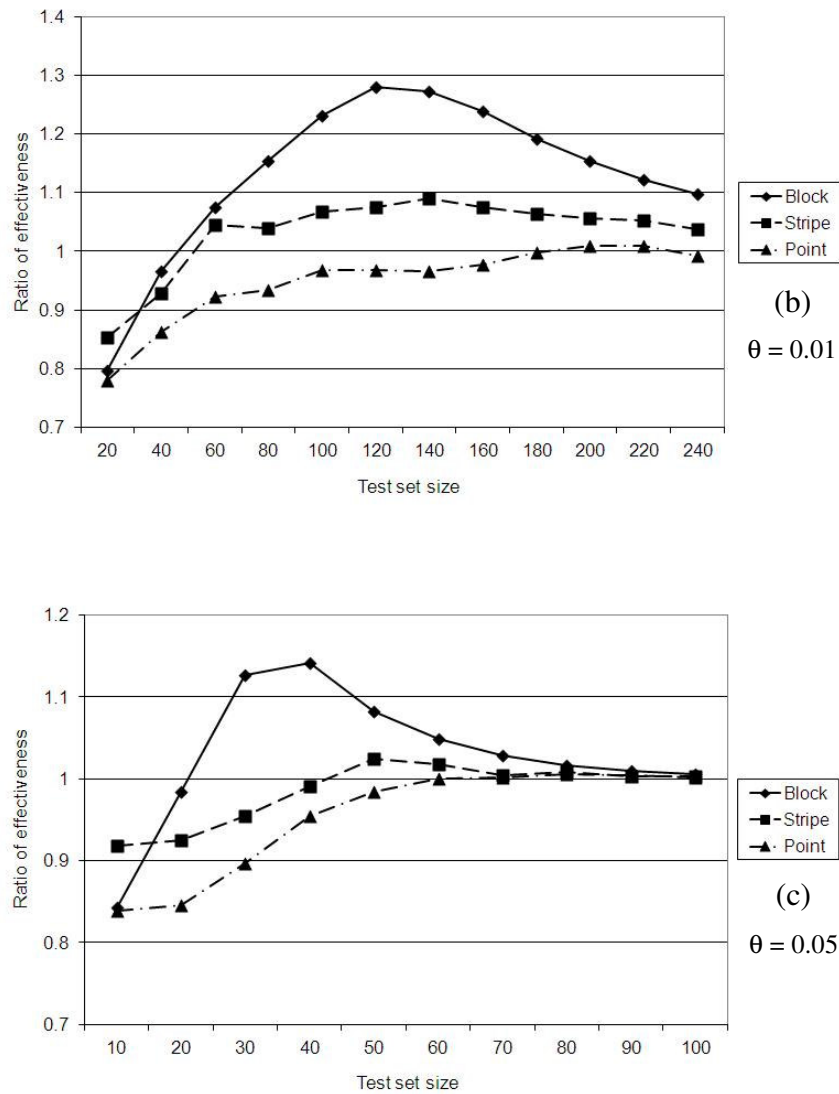


Figura 32. Razão de eficácia entre conjuntos DOTS e RTS

5.3.1. Relação entre a Eficácia dos Conjuntos DOTS e os Padrões de Falha

A análise das Figuras 31 e 32 permite perceber as diferenças de eficácia entre conjuntos de teste gerados aleatoriamente (RTS) e conjuntos de teste orientados a diversidade (DOTS). Fica evidente que as diferenças são mais significativas para falhas padrão tipo bloco, com uma diferença de eficácia máxima da ordem de 35%, obtida no caso

de $\theta = 0.002$ e $|T| = 500$. Para padrões de falha tipo faixa a eficácia dos conjuntos DOTS também é superior aos conjuntos RTS, mas a diferença não é tão expressiva quanto no caso do tipo bloco. Em padrões de falha tipo faixa os conjuntos DOTS são mais eficazes em no máximo 10% (também no caso de $\theta = 0.002$ e $|T| = 500$). Para padrões de falha do tipo ponto os conjuntos RTS mostraram-se mais eficazes que os conjuntos DOTS na maioria das situações. Isto só não ocorre para conjuntos de teste relativamente grandes ($|T| > 60$ no caso de $\theta = 0.05$; $|T| > 200$ no caso de $\theta = 0.01$; e $|T| > 400$ no caso de $\theta = 0.002$).

A Figura 33 mostra a eficácia dos conjuntos RTS e DOTS, $\theta = 0.01$, para o padrão de falha tipo bloco. É mostrada também a distância média entre os dados de teste no conjunto DOTS (linha padrão triangular), valores calculados com a equação a seguir:

$$AvgDist(T) = Div(T) / |T| \quad (5.2)$$

Esta região de falha é um bloco com dimensão 10.0 por 10.0. $Div(T)$ é o valor da diversidade do conjunto de teste T , de tamanho $|T|$. O valor $AvgDist(T)$ representa, portanto, a distância média entre cada dado de teste em T e o seu vizinho mais próximo no domínio D . Quando o valor $AvgDist(T)$ torna-se menor do que 10.0 a eficácia do conjunto de teste DOTS aproxima-se de 1.0 (todos os conjuntos de teste DOTS revelam a região de falha), isto ocorre quando $|T| > 160$. Para $|T| = 160$ a eficácia dos conjuntos de teste RTS fica em 0.8 e continua aumentando, mas sem alcançar eficácia máxima de 1.0, alcançada pelos conjuntos DOTS com $|T| = 160$ (para $|T| = 240$ a eficácia de conjuntos RTS é de cerca de 0.9)

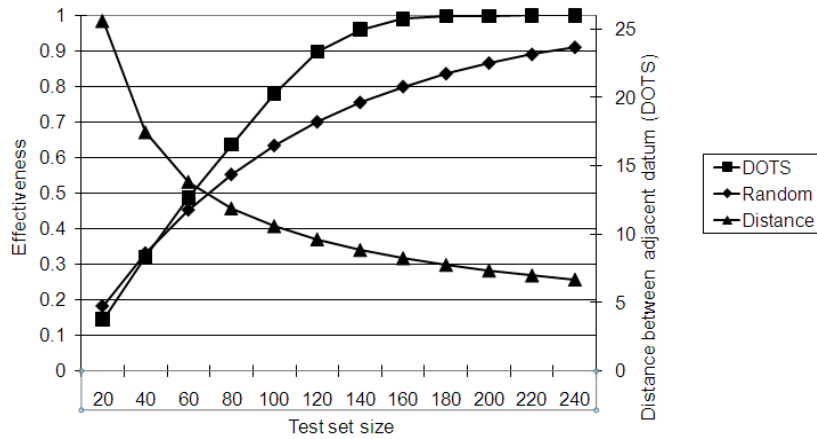


Figura 33. Distância média entre dados de teste do conjunto DOTS e eficácia

Para ilustrar este efeito considere $|T| = 120$, tamanho para o qual a razão de eficácia entre conjuntos DOTS e RTS é máxima. Para este tamanho de conjunto de teste, a diversidade média de conjuntos DOTS, $Div(T) = 1154$, portanto $Div(T)/|T| = 9.62$. Pode-se compreender este efeito fazendo analogia com uma peneira, que impede que “objetos” maiores que um determinado tamanho limite passem na peneira, retendo tais objetos. Neste “efeito crivo” os conjuntos de teste DOTS parecem formar uma “crivo” multidimensional (cuja dimensão é D) cuja granularidade, controlada pelo tamanho do conjunto de teste ($|T|$), define o tamanho presumido das regiões de falha que serão “filtradas” no domínio de entrada do programa.

5.3.2. Relação entre a Eficácia dos Conjuntos DOTS e o Tamanho do Conjunto de Teste

Na análise de conjuntos DOTS foi destacado que estes conjuntos preenchem uniformemente o domínio de entrada do programa e exercitam os limites do domínio. A simulação realizada permitiu observar que para conjuntos de teste “muito pequenos” a técnica tende a alocar uma grande proporção de dados de teste nos limites do domínio. Por exemplo, se o domínio de entrada é um cubo e $|T| = 9$, 8 dados de teste serão posicionados

nos vértices do cubo e um dado de teste será posicionado no interior do cubo, equidistante dos demais dados de teste.

Esta característica, referida como “efeito pelicular” faz com que os conjuntos RTS sejam mais eficazes se o tamanho do conjunto de teste for muito pequeno. Nesta avaliação o fator utilizado para ajustar o tamanho do conjunto de teste ($|T|$) é a taxa de falhas (θ). Por exemplo, para $\theta = 0.01$, um valor coerente para $|T|$ é por volta de 100 (considerando a Equação 5.1, seria esperado que conjuntos RTS encontrassem regiões de falha em 1267 de 2000 tentativas – 63.39% com esses valores de θ e $|T|$). Neste caso $|T|$ ao redor de 10 pode ser considerado pequeno – apenas 9.56 % dos conjuntos RTS encontrariam regiões de falha). Este efeito pelicular tende a reduzir a eficácia de conjuntos DOTS considerando a premissa adotada de que as regiões de falhas estão localizadas em pontos aleatórios no domínio de entrada. Uma premissa de que “as falhas se escondem nos cantos” (Beizer, 1990), não utilizada na simulação, faria com que este efeito atuasse a favor dos conjuntos DOTS ao invés de contra eles.

Para tamanhos de conjuntos de teste muito grandes um “efeito de saturação” ocorre. Por exemplo, para $\theta = 0.01$ e $|T| = 180$ todos os conjuntos DOTS encontram a região de falha (eficácia de 1.0). A partir deste valor de tamanho de conjunto de teste a eficácia dos conjuntos DOTS permanece nesse limite máximo, enquanto os conjuntos RTS ainda apresentam aumento de eficácia com o aumento de $|T|$ (para $\theta = 0.01$ e $|T| = 180$ a eficácia dos RTS é próxima de 0.8). Importante notar que o “efeito de saturação” significa apenas que a *razão de eficácia* entre conjuntos DOTS e RTS diminui com o aumento de $|T|$ acima de certo limite, mas os primeiros conjuntos continuam sendo superiores aos segundos. Considerando conjuntos RTS e $\theta = 0.01$, para atingir eficácia de 0.99 seria necessário um conjunto de teste com aproximadamente 460 dados de teste.

5.3.3. Relação entre a Eficácia dos Conjuntos DOTS e o Tamanho das Regiões de Falha

A razão de eficácia entre os conjuntos DOTS e os RTS aumenta na medida em que a taxa de falhas (θ) reduz; este efeito é mais acentuado para o padrão de falhas tipo bloco. Este comportamento deve-se também ao efeito pelicular, descrito na seção anterior. Para taxas de falhas menores os tamanhos dos conjuntos de teste utilizados são maiores e conjuntos maiores tendem a anular o efeito pelicular, pois é possível alocar dados de teste nos limites do domínio e também preencher com dados de teste o interior do domínio.

A Figura 34 ilustra a variação da razão de eficácia entre conjuntos DOTS e conjuntos RTS para diferentes valores de taxa de falhas (θ). Para cada taxa de falhas são considerados tamanhos de conjuntos de teste coerentes (com chance média de revelar regiões de falha, conforme descrito na Seção 5.4.1): $|T|=500$ para $\theta = 0.002$; $|T|=100$ para $\theta = 0.01$; e $|T|=40$ para $\theta = 0.05$.

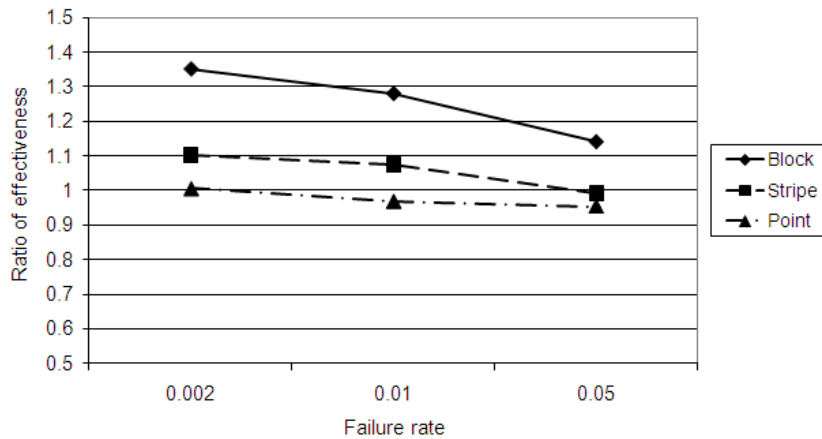


Figura 34. Razão de eficácia e tamanho de região de falha

5.3.4. Simulação: Sumário dos Resultados

Como descrito nas seções anteriores, os resultados da simulação indicam que os conjuntos de teste DOTS podem apresentar uma eficácia maior que a dos conjuntos RTS – um ganho de até 35% – e que este ganho de eficácia parece ser maior para regiões de falha com valores menores de taxas de falhas.

Existe também uma grande influência do tipo de região de falha no desempenho dos conjuntos DOTS. A eficácia desses conjuntos é maior para regiões de falha do tipo bloco, ou do tipo faixa (mais “compactas”) e pior para regiões do tipo ponto.

Em relação ao tamanho do conjunto de teste, a vantagem de eficácia dos conjuntos DOTS sobre os conjuntos RTS tende a ser menor para conjuntos de teste muito pequenos e para conjuntos de teste muito grandes (pequeno ou grande avaliado em relação à taxa de falhas).

Limitações dessa simulação podem ser identificadas: as regiões de falha são aproximações das regiões de falhas geradas por defeitos reais. Estas regiões podem ter formas mais complexas que as consideradas. A simulação considerou um único domínio bidimensional. A consideração de domínios com um número maior de dimensões permitiria considerar também a influência desse fator na eficácia relativa das técnicas.

5.4. Avaliação do Efeito da Diversidade na Cobertura de Fluxo de Dados e no Escore de Mutação

Nesta avaliação um pequeno conjunto de programas foi selecionado para a avaliação da influência de valores de diversidade de conjuntos de teste na eficácia desses conjuntos, avaliada pelo uso das técnicas de análise de fluxo de dados e análise de mutantes (Capítulo 2 e Apêndice A).

Medidas de cobertura alcançada, relativa a critérios baseados em estrutura ou baseados em defeitos, são amplamente utilizadas para a avaliação de qualidade de conjuntos de teste (Wong et al., 1994), (Horgan et al., 1994), (Chen et al., 2001). Conjuntos

de teste que alcançam maior cobertura exercitam o programa de forma mais rigorosa e, portanto, possuem mais chances de revelar os defeitos existentes. Medidas de cobertura do teste são também utilizadas para comparar técnicas de geração de dados de teste. Em geral, se os conjuntos de teste gerados usando uma técnica T1 alcançam valores de cobertura maiores que os conjuntos de teste gerados usando outra técnica T2, é razoável concluir que T1 é mais eficaz que T2. Esse tipo de análise requer que vários conjuntos de teste sejam gerados usando T1 e T2, e que valores médios sejam calculados. Isto porque ocorrem variações nos resultados de cobertura ao se considerar diferentes conjuntos de teste, gerados segundo uma mesma técnica.

Importante notar que medidas de cobertura funcionam como uma avaliação indireta de eficácia. Uma avaliação direta da eficácia de uma técnica de teste (ou de geração de dados de teste) em um contexto determinado (exemplo: uma organização que desenvolve e testa software, ou um laboratório específico de teste de software) requer a aplicação da técnica neste contexto em diversos produtos desenvolvidos, e a avaliação dos defeitos revelados durante o teste e dos defeitos descobertos apenas após a liberação do software, durante o seu uso. As conclusões dessa avaliação seriam referentes ao contexto da organização, mas não necessariamente válidas em outros contextos. Isto é mesmo esta avaliação “ideal” de uma técnica de teste, ainda deveria ser replicada em outros contextos para que conclusões genéricas sobre a eficácia da técnica fossem possíveis.

Uma avaliação extensiva de técnicas de teste, como a avaliação descrita no parágrafo anterior, é desejável, mas dificilmente é possível realizar uma validação inicial de técnica de teste desta forma. Esta dificuldade deve-se ao grande custo e esforço envolvidos e também à natural resistência de organizações em incorporar em suas práticas o uso de novas técnicas recém-desenvolvidas, exemplo: (Shull et al., 2008). Devido a estas dificuldades, tipicamente as técnicas de teste são avaliadas em ambientes controlados e por meio de experimentos. Em particular, o *escore de mutação* (cobertura de mutantes) na técnica Teste de Mutação é considerado como uma estimativa bastante confiável da qualidade dos conjuntos de teste para revelar defeitos reais (Andrews et al., 2005).

5.4.1. Procedimento de Avaliação

Os conjuntos de teste gerados aleatoriamente (RTS) e conjuntos de teste orientados à diversidade (DOTS) foram avaliados por meio de um procedimento que possui três fases: preparação da avaliação, execução da avaliação, e análise de resultados.

Como já destacado, este procedimento visa a avaliar os valores de cobertura alcançados ao se utilizar cada técnica de geração de dados, em diferentes programas, diferentes tamanhos dos conjuntos de teste, e utilizando diferentes técnicas de teste para avaliar a cobertura.

A Figura 35 a seguir, adaptada de Rosenberg (2008), ilustra o contexto do procedimento de avaliação. O “processo/técnica em estudo” refere-se à utilização de conjuntos de teste orientados a diversidade (DOTS), o “processo de medição” consiste na avaliação de valores de cobertura alcançados usando conjuntos de teste DOTS e RTS – procedimento descrito a seguir, o “processo estatístico” diz respeito à avaliação do nível de significância estatística dos resultados, o “processo de decisão” significa elaborar conclusões sobre a eficácia e adequação de uso das técnicas, mas também conclusões sobre a adequação dos processos de medida, e do processo estatístico.

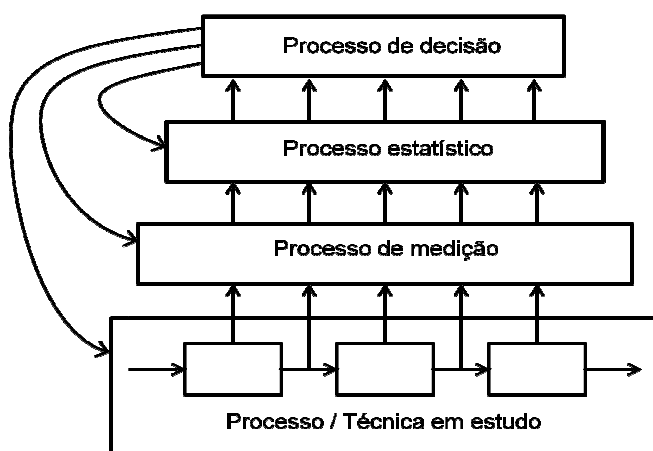


Figura 35. Contexto do procedimento de avaliação

Entradas e saídas do procedimento

Variáveis de entrada do procedimento: programa em teste, número de conjuntos de teste, tamanho do conjunto de teste, técnica de geração dos dados de teste, critério para avaliação de cobertura do teste.

Variáveis de resposta (saída) do procedimento: valores de cobertura alcançada pelos conjuntos de dados de teste com respeito aos critérios Todos os Usos, Todos os Ducaminhos e Análise de Mutantes, valores respectivos de desvio padrão, resultados estatísticos.

Início do Procedimento

Preparação

1. Selecionar um conjunto de programas para avaliação e valores para os tamanhos dos conjuntos de teste (NTD);
2. Para cada programa: identificar a interface, implementar um *driver* que recebe os dados de teste da meta-heurística de Repulsão Simulada (SR-DOTG) e executa-os no programa em teste;
3. Definir o número de conjuntos de teste a serem utilizados (NTS). Para esta avaliação NTS=10;

Execução

Para cada programa, e para cada tamanho de conjunto de teste NTD, executar passos 4 e 5:

4. Gerar NTS conjuntos RTS de tamanho NTD. Cada RTS é gerado com o uso de um gerador uniforme de números aleatórios (geração com reposição), implementado com o uso de funções da linguagem C, e integrado na meta-heurística SR-DOTG;
5. Para cada RTS, utilizar a meta-heurística SR-DOTG para gerar NTS conjuntos DOTS (gerados pela otimização de diversidade dos conjuntos RTS);

Análise

Para cada um dos NTS conjuntos RTS, e para cada um dos NTS conjuntos DOTS, executar passos 6 a 9:

6. Utilizando a ferramenta Poke-Tool, versão *script*, (Chaim, 1991), (Chaim et al., 1995) avaliar os valores de cobertura de fluxo de dados relativos aos critérios de teste Todos-os-Usos e Todos-os-du-caminhos;
7. Utilizando a ferramenta Proteum (Delamaro et al., 1996) avaliar o valor do escore de mutação relativo ao critério Análise de Mutantes.
8. Para cada programa, cada tamanho de conjunto de teste (NTD), e cada critério de teste, calcular o valor médio de diversidade dos conjuntos de teste e os valores médios e o desvio padrão das coberturas alcançadas (considerando os valores obtidos pelos NTS casos de teste de cada técnica de geração de dado – RTS e DOTS).
9. Para cada programa, cada tamanho de conjunto de teste (NTD), e cada critério de teste, realizar a análise estatística dos resultados de cobertura.

Fim do Procedimento

5.4.2. Programas e *Drivers* de Teste

Foram utilizados na avaliação três programas já extensivamente utilizados em análises por outros autores. O programa *tritype* recebe como entrada três variáveis do tipo inteiro que representam os comprimentos dos lados de um triângulo, e classifica o tipo de triângulo e calcula a sua área (Myers, 1979), (Frankl e Weyuker, 1988). O programa *expint* recebe como entradas um variável tipo inteiro (n), uma variável tipo real (x), e calcula a exponencial integral de x (Press et al., 1992). O programa *tcas* é parte de um sistema aéreo de alerta de tráfego e de prevenção à colisão que recebe informações sobre altitudes de voos, alguns parâmetros de ajuste, e produz como saídas alertas de segurança de voo, ex: (Rothermel et al., 1999).

Para testar os programas é necessário implementar um *driver* que recebe valores de entrada (valores dos dados de teste, que formam os conjuntos de teste), sejam eles de

conjuntos RTS ou DOTS. O Capítulo 4 descreve esses drivers e como eles podem ser utilizados para o teste de programas que recebem como entrada estruturas mais complexas.

5.4.3. Conteúdo e Formato dos Resultados

As Tabelas 3, 5, 7 e 9 nas seções seguintes resumem os resultados obtidos com a avaliação realizada por meio do procedimento descrito na Seção 5.5.1. Para cada tamanho de conjunto de teste ($|T|$) são apresentados o valor de diversidade do conjunto de teste (Div), e os valores de cobertura alcançados. Estes valores são: a cobertura alcançada em relação ao critério de fluxo de dados Todos os Usos ($Uses$); a cobertura alcançada em relação ao critério de fluxo de dados Todos os DU-caminhos (DuP) (Maldonado, 1991), (Chaim, 1991); e, o escore de mutação em relação ao critério de teste baseado em defeitos Análise de Mutantes (DeMillo et al., 1978), (Delamaro et al., 1996) (Delamaro et al., 2007).

Os valores de cobertura de fluxo de dados (critério Todos os Usos e critério Todos os DU-caminhos) referem-se ao percentual de elementos requeridos exercitados em relação ao total de elementos requeridos para o programa em teste. Os valores de cobertura para o critério Análise de Mutantes representam o percentual de mutantes mortos em relação a todos os mutantes gerados para o programa em teste.

Todos os valores nas tabelas são obtidos por meio do cálculo de valores médios (referidos por *avg*) de 10 instâncias de casos de teste ($NTS = 10$), para cada tamanho de conjunto de teste ($|T|$), e para cada técnica de geração de dados (RT e DOTG). Para cada valor de cobertura a tabela mostra o valor do desvio padrão da cobertura (referidos por *std*) calculado considerando as 10 instâncias. Notar que em todos os casos é feita uma comparação das técnicas de geração de dados (RT e DOTG) em igualdade de condições (i.e., mesmos valores para $|T|$ e mesmo critério de teste).

Não foi avaliada a não executabilidade de elementos requeridos de critérios baseados em análise de fluxo de dados (Associações definição-uso e DU-caminhos), nem a equivalência de mutantes. Ou seja, considerando os elementos requeridos não exercitados e os mutantes não mortos durante o teste, podem existir elementos requeridos não

executáveis e mutantes equivalentes. A não identificação desses elementos e mutantes não prejudica a avaliação, cujo propósito é o de comparar as duas técnicas de geração de dados de teste (geração aleatória e geração orientada à diversidade).

5.4.4. Análise Estatística

Foi conduzida uma análise dos resultados obtidos no experimento. Esta análise, um teste de significância estatística dos dados, visa a avaliar se há ou não evidências estatisticamente significativas de diferenças entre os valores das variáveis de resposta consideradas no experimento. O objetivo desta análise é obter indicações de se as diferenças entre os valores de cobertura alcançados pelas técnicas RTS e DOTS são estatisticamente significativas ou não, permitindo uma maior confiabilidade acerca dos resultados da comparação de eficácia (Larson e Farber, 2009), (Rosemberg, 2008).

Neste tipo de análise cria-se um modelo do que seria observado se apenas variações ao acaso ocorressem – a *hipótese nula*. Caso as diferenças observadas sejam muito improváveis de ocorrerem apenas pelo acaso, então conclui-se que as duas amostras não são da mesma população, e sim de duas populações com características significativamente diferentes. Rejeita-se neste caso a hipótese nula e aceita-se a hipótese alternativa, que normalmente representa uma alegação que se deseja verificar. Esta conclusão é passível de erro pela própria natureza da análise. Qualquer teste estatístico não pode provar hipóteses de pesquisa, pode apenas apoiá-las ou refutá-las (Larson e Farber, 2009).

A definição do tipo da análise estatística a ser realizada levou em consideração os seguintes pontos relacionados à natureza do experimento (Rosemberg, 2008):

- As variáveis de resposta do experimento são os valores médios de cobertura atingidos pelos conjuntos de dados de teste aplicados na execução dos programas. Deseja-se comparar valores de cobertura obtidos pelos conjuntos gerados aleatoriamente (RTS) com os valores obtidos pelos conjuntos orientados à diversidade (DOTS). Trata-se, portanto, de uma análise estatística de duas amostras,

colhidas de duas populações distintas (populações de todos os RTS, ou de todos os DOTS existentes).

- Como as duas amostras analisadas são de populações distintas e não há correspondência direta entre os conjuntos aleatórios e os conjuntos orientados à diversidade (RTS e DOTS), é razoável presumir que se tratem de amostras independentes.
- O tamanho da amostra é dado pelo número de conjuntos de dados de teste considerados no cálculo de cada valor médio de cobertura. Conforme já destacado este número é igual a dez para ambas as amostras (NTS=10).
- A análise das variáveis de resposta deve ser realizada para cada par de amostras RTS e DOTS, relacionado a uma situação específica avaliada pelo experimento.
- Cada situação específica é caracterizada por um conjunto de valores para as variáveis de entrada do experimento e visa a avaliar estatisticamente os valores de cobertura obtidos na situação. Cada situação, portanto é determinada por valores específicos para: o tamanho do conjunto de dados teste (entre 20 e 150 dados de teste); a técnica de geração dos dados de teste (DOTS ou RTS); o critério para avaliação da cobertura obtida no teste (Todos os Usos, Todos os Du-caminhos e Análise de Mutantes) e o programa em teste (tritype, expint ou tcas).

Com base nesses pontos descritos anteriormente concluiu-se pela realização de um teste de hipóteses paramétrico para diferença entre médias de duas amostras independentes e de tamanho pequeno (menor do que 30). Foi utilizada a estatística de teste t (t-test), com distribuição amostral de Estudante (uma aproximação assintótica da distribuição Normal), com nove graus de liberdade ($gl = n-1$, $n = 10$).

Considerando o objetivo de avaliar se as diferenças entre as médias são significativas estatisticamente optou-se pelo teste de hipóteses bicaudal com as seguintes hipóteses:

- Hipótese nula (H_0): $\mu_{RTS} = \mu_{DOTS}$
- Hipótese alternativa (H_1): $\mu_{RTS} \neq \mu_{DOTS}$

A Hipótese nula significa que não há diferença estatisticamente significativa entre as médias de cobertura obtidas pelas amostras de conjuntos de teste RTS (μ_{RTS}) e DOTS (μ_{DOTS}). A hipótese alternativa, ao contrário, significa que há diferença estatisticamente significativa entre as médias de cobertura obtidas pelas amostras de conjuntos de teste RTS (μ_{RTS}) e DOTS (μ_{DOTS}). Isto pode ocorrer com $\mu_{\text{RTS}} < \mu_{\text{DOTS}}$ (conjuntos DOTS mais eficazes) ou com $\mu_{\text{RTS}} > \mu_{\text{DOTS}}$ (conjuntos RTS mais eficazes).

Como a análise é feita sobre amostras de conjuntos de dados de teste, e não sobre toda a população dos possíveis conjuntos de dados de teste, há a possibilidade da ocorrência de erros no teste de hipóteses. O Erro Tipo I, também chamado de Falso-positivo, significa rejeitar a hipótese nula quando ela é verdadeira, é de especial importância. Cometer este erro equivale a afirmar sobre a superioridade de uma das técnicas de teste quando, na verdade, elas são equivalentes. Por exemplo, concluir que os conjuntos DOTS são mais eficazes que os conjuntos RTS (ou vice-versa) quando isto não é verdadeiro.

O nível de significância de um teste de hipóteses é a probabilidade máxima permitida de ocorrer um Erro Tipo I e é denotado por α . Um valor pequeno para α significa que se deseja uma pequena probabilidade de rejeitar uma hipótese nula verdadeira, tipicamente α assume valores como 0,10, 0,05 e 0,01. Por exemplo $\alpha = 0,01$ significa que apenas em uma de 100 vezes a hipótese nula será rejeitada sendo ela verdadeira. Em experimentos em engenharia de software um valor frequentemente usado é 0,05, mas alguns trabalhos adotam valores ainda mais conservadores. Por exemplo, (Frankl e Weiss, 1993) usam $\alpha = 0,01$ em um teste de hipóteses para avaliar a significância na diferença de eficácia entre critérios baseados em fluxos de dados em relação à cobertura de ramos.

Conforme destacado, normalmente um teste de hipóteses resulta nos resultados: H_0 rejeitado, ou H_0 não rejeitado, considerando um valor α definido. Uma informação mais detalhada sobre a significância dos resultados pode ser obtida pela determinação e avaliação dos valores p (p -value). Estes valores representam a probabilidade de erro envolvida em aceitar a hipótese alternativa (H_1), indicando a existência de diferença significativa entre as técnicas (DOTS e RTS), e rejeitando H_0 . Quanto menor for o valor p

do teste, mais forte é a evidência contra a hipótese nula e, portanto, mais significativo é o resultado do ponto de vista estatístico. A análise dos valores p obtidos segue diretrizes em (Rosemberg, 2008), que relaciona esses valores com o nível de evidência contra a hipótese nula (H_0):

- $p < 0.01$: evidência muito forte contra H_0 .
- $0.01 \leq p < 0.05$: evidência forte contra H_0 .
- $0.05 \leq p < 0.10$: evidência moderada contra H_0 .
- $p \geq 0.10$: pouca ou nenhuma evidência contra H_0 .

Valores menores de p aumentam a evidência contra a hipótese nula e, portanto, aumentam a confiança de que uma das técnicas de teste comparadas (RTS e DOTS) é superior à outra em relação aos valores de cobertura obtidos. Neste caso, qual técnica é superior fica evidente, visto que a técnica mais eficaz apresenta valores significativamente superiores para a cobertura média.

Na seção seguinte, para cada tabela com resultados de cobertura é apresentada uma tabela associada com informações estatísticas sobre os resultados. A análise estatística é realizada para cada par de valores de cobertura, alcançados pelos conjuntos RTS e DOTS, para cada critério de teste (Todos os Usos e Todos os DU-caminhos, Análise de Mutantes), e para cada tamanho de conjunto de teste ($|T|$). Notar que todos os resultados da análise estatística referem-se a resultados produzidos em condições específicas (i.e., um valor para $|T|$ e um critério de teste) ¹².

5.4.5. Resultados da Avaliação

Nesta seção são apresentados os resultados da avaliação descrita anteriormente. Para cada programa utilizado no experimento é feita uma breve descrição da interface do

¹² Existem diversos pacotes de análise estatística “livres” disponíveis. Por exemplo, é possível realizar o t-test nos sítios – consulta em Junho de 2012: (<http://www.usablestats.com/calcs/2samplet>), (<http://www.graphpad.com/quickcalcs/ttest1.cfm>), (<http://www.danielsoper.com/statcalc3/default.aspx>)

programa e da sua funcionalidade, são apresentadas tabelas com os resultados relativos à cobertura do teste e tabelas com resultados das respectivas análises estatísticas.

Programa expint

O programa expint recebe duas variáveis: uma do tipo inteiro (n) e uma do tipo real (x). Para definir faixas de valores para as variáveis de entrada (limites inferiores e superiores) foi realizada uma breve análise da especificação e do código do programa. Este tipo de análise procura identificar faixas de valores amplos o suficiente para cada variável de forma que o programa possa ser exercitado adequadamente (valores muito restritos podem tornar partes do código fonte, ou funcionalidades, não executáveis – este efeito ocorreria equivalentemente para DOTS e RTS). Foram definidas as faixas de valores: n de 0 a 120 e x de 0.0 a 120.0. Um driver foi utilizado para mapear valores da variável x de 0.0 a 120.0 (como tratadas pela meta-heurística SR-DOTG) para valores de 0.0 a 1.20, pela divisão por 100, passando os resultados como entrada para o programa em teste.

Os elementos requeridos exercitados usando conjuntos de teste DOTS, mas não exercitados por conjuntos RTS são relacionados principalmente a partes do programa que calculam valores de saída quando a entrada $x \leq 1$. Esta parte do código apresenta um laço, com dois comandos de seleção internos, cujas condições são influenciadas indiretamente pelos valores de entradas atribuídos à variável x. A dispersão de valores de x parece ser responsável pela cobertura adicional de alguns elementos requeridos pelos critérios de teste. Além disto, verificou-se também que elementos requeridos relacionados a condições extremas (tais como $n=0$ e $x=0$) foram exercitados mais por conjuntos de teste DOTS do que pelos conjuntos RTS.

A Tabela 3 mostra os resultados de cobertura obtidos pelos conjuntos de teste DOTS e pelos conjuntos RTS. A Tabela 4 apresenta informações estatísticas (valores p) associadas aos valores de cobertura. Conforme descrito nas seções anteriores, cada valor p refere-se a um par de valores de cobertura no programa expint (alcançados pelos conjuntos RTS e pelos conjuntos DOTS), para cada critério de teste (*Uses*, *DuP* e *Mut*) e para cada tamanho do conjunto de teste (|T|).

Nas tabelas de resultados estatísticos (Tabelas 4, 6, 8 e 10) são destacados em **negrito** os valores p relacionados a evidências estatísticas fortes, ou muito fortes ($p < 0.05$). Os resultados estatísticos referentes a todos os programas são discutidos na Seção 5.5.6.

Tabela 3. expint - resultados de cobertura para DOTS e RTS

T	Random Test Sets								Diversity-Oriented Test Sets							
	Div	Uses		DuP		Mut		Div	Uses		DuP		Mut		Div	Mut
		avg	std	avg	std	avg	std		avg	std	avg	std	avg	std		
20	321.7	74.3	8.57	52.1	4.41	0.69	0.11	635.1	71.6	0	53.5	0	0.72	0.0005		
50	409.6	82.1	1.77	58.1	2.91	0.71	0.03	1217.8	83.8	0.51	57.7	0.73	0.83	0.0005		
100	628.5	82.7	1.23	58.9	1.43	0.75	0.11	1374.3	88.8	0.50	62.8	0.72	0.99	0.0001		
150	766.6	83.8	1.34	60.3	1.84	0.81	0.12	1510.7	89.2	0	63.4	0	0.99	0		

Tabela 4. Resultados estatísticos (valores p) para expint

T	Uses	DuP	Mut
20	0.186410	0.167660	0.046730
50	0.117420	0.127310	0.008730
100	0.000250	0.000330	0.000260
150	0.000250	0.000500	0.000250

Programa tritype

O programa tritype, extensamente utilizado em avaliações na área de teste de software, recebe como entrada três valores inteiros não negativos que representam tamanhos dos lados de um triângulo e que devem estar ordenados de forma decrescente. No experimento a interface consiste de três valores inteiros a, b, c que variam de 0 a 50. O programa produz como saída a classificação do triângulo (retângulo, agudo, obtuso, equilátero e isósceles) e o valor da sua área, caso contrário, identifica que os lados não formam um triângulo, ou identifica valores não ordenados na entrada.

Neste programa o fato dos conjuntos DOTS combinarem faixas de valores para os comprimentos dos lados dos triângulos (a, b, c), torna tais conjuntos de teste mais propensos a exercitarem elementos requeridos associados ao triângulo agudo e também ao triângulo obtuso. Os conjuntos RTS com certa frequência falham em exercitar elementos requeridos associados a um desses dois tipos de triângulo em tentativas aleatórias mal sucedidas. A combinação de valores extremos dos conjuntos DOTS exercita também

elementos requeridos associados à identificação de triângulos inválidos (ex: 0,0,0 ou 50,0,0). No entanto, tanto os conjuntos DOTS quanto os conjuntos RTS falham na maioria das vezes em exercitar elementos requeridos associados a triângulos retângulos, que representam uma porção muito pequena do domínio de entrada do programa (dadas as restrições estabelecidas no código e para este domínio de entrada, existe apenas um dado de teste que exercita esses elementos requeridos $a = 5$, $b = 4$, $c = 3$). A Tabela 5 mostra os resultados de cobertura alcançados pelos conjuntos DOTS e pelos conjuntos RTS. A Tabela 6 apresenta informação estatística dos resultados (valores p).

Tabela 5. tritype resultados de cobertura para DOTS e RTS

$ T $	Random Test Sets							Diversity-Oriented Test Sets						
	Div	Uses		DuP		Mut		Div	Uses		DuP		Mut	
		avg	std	avg	std	avg	std		avg	std	avg	std	avg	std
30	305.1	47.2	21.54	44.9	19.20	0.43	0.35	599.9	57.9	7.67	50.7	7.49	0.53	0.12
64	495.3	63.1	10.0	61.4	8.87	0.54	0.23	922.4	83.7	6.28	76.4	8.45	0.65	0.08
100	627.9	70.7	9.37	68.9	8.75	0.65	0.11	1194.9	83.7	5.76	77.4	7.90	0.79	0.08
150	832.1	70.7	8.22	68.9	7.23	0.65	0.10	1538.8	83.7	3.02	77.4	5.02	0.85	0.06

Tabela 6. Resultados estatísticos (valores p) para tritype

$ T $	Uses	DuP	Mut
30	0.064200	0.188960	0.024820
64	0.000280	0.000860	0.000260
100	0.001405	0.060355	0.000475
150	0.001243	0.034002	0.000310

Programa tcas

O programa tcas é um módulo integrante do software TCAS, um sistema de segurança de voo presente em aeronaves com o objetivo de evitar colisões aéreas com outras aeronaves. Este módulo foi utilizado em trabalhos para a avaliação de técnicas de teste e de técnicas de minimização de conjuntos de teste (Rothermel et al., 1999). Informações de voo da aeronave e de aeronaves vizinhas, a uma determinada distância limite, são utilizadas para projetar a posição futura de cada aeronave e emitir mensagens. Essas mensagens representam diferentes níveis de alerta e, eventualmente, sugerem manobras para evitar colisões.

O programa tcas recebe como entradas 15 variáveis. Seis variáveis inteiras se referem a valores absolutos de altitudes para aviões e a altitudes relativas que os separam. Cinco variáveis lógicas determinam o tipo de resposta do algoritmo a partir da análise das altitudes. Uma variável de entrada tipo *flag* pode assumir valores 0,1,2 e 3; cada valor indica um nível de altitude distinto da aeronave, que implica em diferentes mensagens. Os principais procedimentos do tcas são o principal (main) e um denominado alt-sep-test, que realiza análises de diferenças de altitudes e é chamado pelo procedimento principal (o procedimento alt-sep-test, por sua vez, chama 6 outros procedimentos menores).

Para este programa decidiu-se ajustar os valores de altitude (relativos às seis variáveis inteiras) variando entre de 0 a 800. As outras variáveis (cinco variáveis lógicas e uma do tipo *flag*) são representadas também como valores inteiros variando de 0 a 800. Um driver de teste decodifica as entradas recebidas deste domínio de busca da meta-heurística SR-DOTG para o domínio de entrada do programa tcas. No caso de valores de altitude o mapeamento é direto, isto é, o driver recebe o valor da SR-DOTG e o repassa ao programa tcas. No caso de valores lógicos o driver recebe os valores que variam de 0 a 800 e realiza um mapeamento para valores lógicos (0 ou 1). Valores entre 0 e 400 são mapeados para o valor 0 e valores entre 400 e 800 são mapeados para o valor 1. O mapeamento para variável *flag* é feita de forma análoga, isto é o intervalo (0, 800) é dividido em 4 faixas de valores, cada faixa mapeada por um valor *flag*: entre 0 e 200 mapeado para valor 0, entre 200 e 400 mapeado para valor 1, entre 400 e 600 mapeado para valor 2, entre 600 e 800 mapeado para valor 3.

Este tipo de mapeamento faz com que a diversidade dos conjuntos DOTS gerados pela meta-heurística SR-DOTG seja transferida para o domínio de entrada do programa em teste. Mais precisamente, o efeito do mapeamento é o de transformar a diversidade de valores existentes nos conjuntos DOTS em uma diversidade de classes de valores, existente no domínio de entrada do programa em teste. Em relação aos valores usados nos dados de teste o efeito é o de realizar a combinação de classes de valores, ao invés de realizar a combinação dos próprios valores.

Notar que o mapeamento, feito por um driver, de uma faixa de valores de entrada para valores de saída oferece flexibilidade na utilização dos conjuntos DOTS. Por exemplo, uma faixa de valores de entrada (ex: de 0 a 100) pode ser mapeada em 4 saídas (1,2,3,4) e estas saídas podem funcionar como índice de um vetor – supondo um vetor pag que contém opções de pagamento em uma transação, cada valor (1 ... 4) pode definir uma opção: pag[1]: cartão débito, pag[2]: cartão crédito, pag[3]: cheque, etc. Outro recuso é mapear uma faixa de valores de entrada para valores de saída de forma não uniforme. Por exemplo, uma faixa de valores de entrada (ex: de 0 a 100) pode ser mapeada para dois valores de saída (0,1) de modo que a saída seja 0 para entradas entre 0 e 40 e 1 para entradas entre 40 e 100. Neste caso o valor 1 seria privilegiado.

Os resultados para o programa tcas são descritos nas Tabelas 7 e 9. A Tabela 7 mostra os resultados de cobertura para o procedimento main e os valores de diversidade dos conjuntos de teste. A Tabela 9 mostra os valores de cobertura para o procedimento alt-sep-test e o escore de mutação relativos a ambos os procedimentos (main e alt-sep-test). As Tabelas 8 e 10 mostram informação estatística (valores p).

Para este programa os resultados de cobertura para o procedimento main com o uso dos conjuntos de teste DOTS foram abaixo do esperado, considerando os resultados obtidos em outros programas. Os conjuntos DOTS apresentam eficácia menor do que conjuntos RTS para $T = 100$ e $T = 150$. Os resultados de cobertura para alt-sep-test indicam uma melhor eficácia dos conjuntos DOTS comparados aos conjuntos RTS considerando os critérios Todos os Usos e Todos os Du-caminhos. Os resultados relacionados ao critério Análise de Mutantes relativos a ambos os procedimentos (main e alt-sep-test) indicam uma pequena vantagem em eficácia dos conjuntos DOTS sobre os conjuntos RTS.

Tabela 7. tcas-main resultados de cobertura para DOTS e RTS

T	Random Test Sets					Diversity-Oriented Test Sets				
	Div	Uses		DuP		Div	Uses		DuP	
		avg	std	avg	std		avg	std	avg	std
50	34053	95.1	0	41.9	1.62	51282	95.1	0	42.1	1.42
100	63704	95.1	0	58.5	1.12	92311	95.1	0	57.6	1.11
150	91135	95.1	0	68.1	2.28	119068	95.1	0	67.0	2.14

Tabela 8. Resultados estatísticos (valores p) para tcas-main

$ T $	<i>Uses</i>	<i>DuP</i>
50	0.171965	0.377385
100	0.171965	0.058450
150	0.171965	0.146230

Tabela 9. Resultados de cobertura para alt-sep-test e escore de mutação para main e alt-sep-test

$ T $	Random Test Sets						Diversity-Oriented Test Sets					
	<i>Uses</i>		<i>DuP</i>		<i>Mut</i>		<i>Uses</i>		<i>DuP</i>		<i>Mut</i>	
	<i>avg</i>	<i>std</i>	<i>avg</i>	<i>std</i>	<i>avg</i>	<i>std</i>	<i>avg</i>	<i>std</i>	<i>avg</i>	<i>std</i>	<i>avg</i>	<i>std</i>
50	66.3	5.09	54.9	8.05	0.583	0.081	72.6	5.07	65.0	7.45	0.625	0.044
100	72.6	3.33	64.9	5.27	0.616	0.034	75.8	3.02	69.9	5.23	0.659	0.021
150	72.6	5.98	64.9	9.46	0.677	0.037	76.8	4.12	71.6	8.25	0.702	0.011

Tabela 10. Resultados estatísticos (valores p) para alt-sep-test, main

$ T $	<i>Uses</i>	<i>DuP</i>	<i>Mut</i>
50	0.032420	0.010700	0.012025
100	0.045620	0.000310	0.009055
150	0.085065	0.085040	0.002795

O desempenho dos conjuntos de teste DOTS abaixo do esperado para o programa tcas foi aparentemente uma consequência da definição de tamanhos de conjuntos de teste menores do que os necessários para uma cobertura adequada do domínio de entrada do programa. Como o número de dimensões deste domínio (definida pelo número de variáveis de entrada) é maior do que no caso dos programas anteriores o domínio de entrada é expressivamente maior, requerendo uma quantidade substancialmente maior de dados de teste para a cobertura completa deste domínio. O indício de que este fato ocorre vem da observação de que para este programa os conjuntos DOTS exploram os pontos extremos do domínio de entrada, mas deixam a parte interior deste domínio com menos pontos (dados) de teste do que os conjuntos RTS de mesmo tamanho.

5.4.6. Cobertura de Fluxo de Dados e Escore de Mutação: Sumário dos Resultados

A Tabela 11 resume os resultados encontrados nas tabelas anteriores (Tabelas 3, 5, 7 e 9 relativas aos programas expint, tritype e tcas, respectivamente) e mostra a eficácia dos

conjuntos DOTS comparada à eficácia dos conjuntos RTS (conforme descrito na Seção 5.5, os valores de cobertura são identificados como indicadores de eficácia). Cada valor na tabela foi obtido por meio do seguinte procedimento de dois passos:

- i. É calculada a *Razão-Programa-Critério-Tamanho* que representa a razão entre a cobertura alcançada pelos conjuntos DOTS e a cobertura alcançada pelos conjuntos RTS para cada programa, critério de teste, e tamanho do conjunto de teste;
- ii. É calculada a *Razão-Programa-Critério* que representa os valores médios para a *Razão-Programa-Critério-Tamanho* (calculada no passo i) considerando todos tamanhos de conjuntos de teste, para cada programa e critério de teste.

Notar que *Razão-Programa-Critério-Tamanho* calculada no passo (i) mede o quão melhor (ou o quão pior) foram os conjuntos DOTS comparados aos conjuntos RTS em uma situação específica (um dado programa, critério de teste e tamanho do conjunto de teste). No passo (ii) é calculada a *Razão-Programa-Critério* que resume esta informação pela abstração dos tamanhos dos conjuntos de teste.

A Tabela 11 mostra que os conjuntos DOTS foram mais eficazes que os conjuntos RTS em 10 do total de 12 situações. Nessas 10 situações os conjuntos DOTS apresentam razão de eficácia que varia de 1.032 (programa expint avaliado com o critério Todos os Usos) até 1.230 (programa tritype avaliado com o critério Análise de Mutantes).

Outra característica importante dos conjuntos DOTS pode ser percebida pela análise dos valores de desvio padrão dos valores de cobertura alcançados no teste – Tabelas 3, 5, 7 e 9. Os valores de desvio padrão da cobertura para os conjuntos DOTS são consistentemente menores que os valores de desvio padrão da cobertura para os conjuntos RTS quando se comparam situações similares (um programa específico, um critério de teste, e um tamanho de conjunto de teste). Esta característica pode ser vista como positiva. Um menor valor de desvio padrão na eficácia de uma técnica indica uma menor variabilidade esperada da eficácia da técnica ao se considerar vários diferentes conjuntos de teste. Isto tende a tornar os resultados do teste mais constantes (e, portanto, mais previsíveis) ao se usar conjuntos DOTS no teste, comparado ao uso dos conjuntos RTS.

Tabela 11. Razão entre a cobertura de conjuntos DOTS e conjuntos RTS

Program	Uses	DuP	Mut
<i>Expint</i>	1.032	1.035	1.189
<i>Tritype</i>	1.227	1.115	1.230
<i>Tcas-main</i>	1.000	0.989	1.059
<i>tcas-alt-set-test</i>	1.064	1.118	1.059

A análise detalhada das Tabelas 3, 5, 7 e 9 mostra um total de 39 pares de valores de cobertura (alcançados por conjuntos RTS e por conjuntos DOTS) para todos os programas, critérios de teste, e tamanhos de conjunto de teste. Desse total de 39 pares de valores, em 35 pares a cobertura alcançada pelos conjuntos DOTS foi maior do que a cobertura alcançada pelos conjuntos RTS ($cov(DOTS) \geq cov(RTS)$). Em apenas 4 pares de valores a cobertura alcançada pelos conjuntos DOTS foi menor do que a cobertura alcançada pelos conjuntos RTS ($cov(DOTS) < cov(RTS)$)¹³.

Em relação aos resultados estatísticos, como destacado na Seção 5.5.4, p-valores menores indicam evidências mais fortes contra H_0 e, portanto, fornecem uma indicação mais segura de que uma técnica é realmente mais eficaz do que a outra (Larson e Farber, 2009). Para analisar as tabelas de informações estatísticas (Tabelas 4, 6, 8 e 10) é necessário inicialmente distinguir duas classes de resultados: a classe (a) é caracterizada por $cov(DOTS) \geq cov(RTS)$ e a classe (b) é caracterizada por $cov(DOTS) < cov(RTS)$.

a) $cov(DOTS) \geq cov(RTS)$. Nível de evidência estatística de que os conjuntos de teste DOTS são mais eficazes que os conjuntos de teste RTS. Desses 35 pares de valores de cobertura (alcançados por conjuntos RTS e por conjuntos DOTS), para 17 pares as evidências são muito fortes ($p < 0.01$), em 7 pares as evidências são fortes ($0.01 \leq p < 0.05$), em 4 pares as evidências são moderadas ($0.05 \leq p < 0.10$), e para 7 pares as evidências são pequenas ($p > 0.10$).

b) $cov(DOTS) < cov(RTS)$. Nível de evidência estatística de que os conjuntos de teste RTS são mais eficazes que os conjuntos de teste DOTS. Desses 4 pares de valores de cobertura (alcançados por conjuntos RTS e por conjuntos DOTS), para um par

¹³ Estas situações são: expint, T = 20, critério = uses; expint, T = 50, critério = DuP; tcas-main, T = 100, critério = DuP; e tcas-main, T = 150, critério = DuP.

as evidências são moderadas ($0.05 \leq p < 0.10$), e para 3 pares as evidências são pequenas ($p > 0.10$).

De maneira geral os resultados estatísticos mostram que os conjuntos DOTS são mais eficazes que os conjuntos RTS, com evidências muito fortes, ou com evidências fortes ($p < 0.05$) em 24 pares de valores de cobertura. Por outro lado, em apenas um par de valores de cobertura os conjuntos RTS são mais eficazes do que os conjuntos DOTS, com evidências moderadas ($0.05 \leq p < 0.10$).

5.5. Considerações Finais

Como já destacado anteriormente neste capítulo, avaliações empíricas em engenharia de software, mesmo bem embasadas e realizadas de forma sistemática, apresentam normalmente *ameaças de validade*. No caso dos experimentos conduzidos, pode-se identificar as seguintes ameaças:

- As regiões de falha utilizadas nas simulações são aproximações das regiões de falhas geradas por defeitos reais. Esses tipos de regiões têm sido utilizados para avaliar técnicas de teste (veja Seção 4.5.3). Estas regiões, contudo, podem ter formas mais complexas que as consideradas (fato já destacado na Seção 5.4.4). O estudo mais aprofundado dessas regiões é incipiente (Schneckenburger e Mayer, 2007); avanços nessa área podem levar à necessidade de considerar regiões de falhas adicionais em simulações.
- Foram utilizados apenas programas simples na avaliação de cobertura alcançada por DOTS e por RTS. Mesmo considerando que esses programas são comumente utilizados por outros autores em avaliações, esta limitação dificulta traçar hipóteses de que a maior eficácia dos DOTS sobre os RTS mantêm-se para programas mais complexos. Isto é, o impacto do nível de complexidade dos programas na eficácia relativa das técnicas não foi avaliado.
- O tamanho da amostra de programas é pequeno. Tendo em vista a necessidade de considerar múltiplos fatores na avaliação de cobertura (tamanho do conjunto de

teste, critério de avaliação de cobertura e técnica de geração de dados), e considerando que cada combinação desses parâmetros requer várias instâncias (para tratar a aleatoriedade e calcular valores médios), decidiu-se por limitar o número de programas utilizados na avaliação, para torná-la viável e permitir uma avaliação abrangente em relação aos fatores destacados. Embora esse tipo de estudo seja normalmente feito com amostras pequenas, esta limitação restringe a capacidade de se descobrir características de programas que afetem a eficácia relativa das técnicas.

Essas ameaças de validade sugerem a necessidade de mais experimentos de avaliação da técnica DOTG-ID, para que se ganhe mais confiança sobre a escalabilidade da técnica para programas mais complexos. Contudo, tratou-se de uma avaliação sistemática, abalizada em trabalhos de engenharia de software empírica, ex: (Shull et al., 2008). Esta avaliação também satisfaz os requisitos de suficiência definidos por Ali et al. (2010) para trabalhos de geração de dados baseada em busca (SBST): “oferecer evidências acreditáveis sobre custo e eficácia”, o que inclui levar em conta variações aleatórias e estabelecer comparação com outra técnica; e “oferecer avaliação de confiança dos resultados do ponto de vista estatístico”.

Em uma primeira etapa de avaliação, as meta-heurísticas (SA-DOTG, GA-DOTG e SR-DOTG) foram utilizadas para gerar conjuntos DOTS para diferentes situações, criadas pela variação de valores de parâmetros como: tipos de variáveis; números de variáveis; e tamanhos de conjunto de teste. Além da avaliação qualitativa, uma situação concreta permitiu constatar que a SR-DOTG alcança níveis mais altos de diversidade, requerendo um número de iterações expressivamente inferior às demais.

A SR-DOTG foi selecionada para as avaliações seguintes: uma simulação e a aplicação para o teste de programas selecionados.

Os resultados da simulação realizada mostram que, em geral, conjuntos de teste orientados à diversidade (DOTS) são mais eficazes que conjuntos de teste gerados aleatoriamente (RTS). A vantagem dos DOTS é significativa para regiões de falhas mais

compactas, do tipo bloco e do tipo faixa. Aparentemente, a diversidade não melhora a eficácia para regiões de falha pontuais. Este resultado sugere que a diversidade dos dados de teste é mais importante do que a aleatoriedade desses dados, considerando o objetivo de encontrar as regiões de falha que sejam razoavelmente compactas.

Os resultados em relação à cobertura de fluxo de dados e de escore de mutação também apontam uma superioridade dos conjuntos DOTS em relação aos RTS. Conjuntos de teste que alcançam maiores valores de cobertura de fluxo de dados exercitam de forma mais rigorosa o programa e tendem a ser mais eficazes (Wong et al., 1994), (Horgan et al., 1994). Conjuntos de teste que alcançam maiores valores de escore de mutação também tendem a ser mais eficazes. Esta medida é vista como uma boa estimativa de eficácia do conjunto de teste em experimentos sobre técnicas de teste. (Andrews et al., 2005).

Como descrito na seção anterior, os conjuntos DOTS foram mais eficazes que os conjuntos RTS em 10 do total de 12 situações. Considerando cada par de valores de cobertura, do total de 39 pares de valores, em 35 pares a cobertura alcançada pelos conjuntos DOTS foi maior do que a cobertura alcançada pelos conjuntos RTS. Estatisticamente, as evidências desta superioridade são muito fortes ($p < 0.01$), ou fortes ($0.01 \leq p < 0.05$) em 24 pares de valores de cobertura desses 35 pares. A análise estatística foi obtida por um teste de hipóteses paramétrico para diferença entre médias de duas amostras independentes (t-test), (Larson e Farber, 2009), (Rosemberg, 2008).

Por fim, cabe destacar que a avaliação empírica realizada teve como objeto de estudo a perspectiva do domínio de entrada da diversidade (DOTG-ID) e de conjuntos de alta diversidade sob esta perspectiva (DOTS). A ideia de que a diversidade do teste pode ter papel importante para a eficácia considerando as várias diferentes perspectivas foi proposta no Capítulo 4. Este capítulo não teve o objetivo de avaliar empiricamente esta hipótese, apenas uma perspectiva da diversidade (DOTG-ID) foi desenvolvida e avaliada.

Capítulo 6 – Conclusão

“Our knowledge can only be finite, while our ignorance must necessarily be infinite.”

Karl Popper

Este capítulo apresenta a conclusão do trabalho. É feito um breve resumo, são destacadas as contribuições do trabalho e são descritos trabalhos futuros relacionados à tese.

6.1. Resumo do trabalho

Técnicas e critérios de teste estabelecem elementos requeridos a serem exercitados no teste (Beizer, 1990), por exemplo: exercitar classes de funcionalidades; elementos do código fonte; ou defeitos específicos.

A geração de dados de teste visa selecionar dados de teste do domínio de entrada do programa para satisfazer um critério. É necessário identificar os elementos requeridos pelo critério e as condições necessárias para que estes elementos sejam exercitados.

Esforços têm sido conduzidos no sentido de automatizar a geração de dados de teste. Uma linha de trabalho é a utilização de meta-heurísticas para a geração de dados de teste, área referida como Teste de Software Baseado em Busca – SBST (Harman et. al., 2009), (McMinn. 2011). A ideia geral desses trabalhos é utilizar meta-heurísticas para buscar, no espaço de possíveis entradas do programa, aquelas que satisfaçam um determinado critério.

Esta busca é guiada por uma função de ajuste que avalia dados de teste, ou conjuntos de teste, em relação aos requisitos estabelecidos pelo critério de teste.

O Capítulo 3 apresentou trabalhos que tratam o problema da geração de dados de teste, com ênfase no SBST. Foi percebida uma lacuna em trabalhos nessa área: a falta da identificação de aspectos comuns a serem considerados por engenheiros de software que querem tratar um problema de teste utilizando meta-heurísticas como ferramenta. Em resposta a esta lacuna, foram elaborados: um guia de passos a serem seguidos para se conceber e se estabelecer uma nova abordagem SBST; e um procedimento genérico para a geração de dados com o uso de meta-heurísticas.

Foi proposta uma nova técnica de geração de dados de teste, a Geração de Dados de Teste Orientada à Diversidade (*Diversity Oriented Test Data Generation – DOTG*). Um ponto que distingue esta técnica das demais é que, ao invés de ter como objetivo satisfazer critérios de teste existentes, o alvo da geração de dados é promover uma propriedade do conjunto de teste: a diversidade de dados de teste no conjunto.

A inspiração inicial para esta ideia foi a intuição de que a variedade dos dados de teste tem um papel relevante para a qualidade do teste. Na ausência de informação sobre a localização dos defeitos no código, parece natural que diversificar o teste leve a uma avaliação mais global e confiável do software.

Esta intuição ganhou apoio em trabalhos da área de confiabilidade de software, os quais indicam que estimativas de confiabilidade são melhores (mais precisas) quando se considera “como” os dados de teste exercitam, ao invés de considerar apenas “quantos” dados de teste foram utilizados, ou “quanto tempo” o software foi executado (Whittaker e Voas, 2000). Executar repetidamente o mesmo dado de teste não repercute em aumento da confiabilidade estimada, este é o teste com diversidade zero. Ao contrário, diversificar dados de teste favorece o ganho de conhecimento sobre o software, levando a estimativas mais precisas de confiabilidade.

Trabalhos analíticos sobre o teste baseado em particionamento também apoiam a ideia da diversidade. Gutjahr (1999) propõe um modelo que leva em conta a incerteza sobre

o número de dados que causam falha em um subdomínio, isto é, o autor trata probabilisticamente os valores para as taxas de falhas dos subdomínios – Weyuker e Jeng, (1991) tratam estes valores de forma determinística. O autor pondera que, como os testadores normalmente não sabem onde estão os defeitos, considerar esta incerteza é uma premissa realista. Este modelo indica uma ampla vantagem da estratégia de se diversificar a alocação de dados de teste, implícita no teste baseado em particionamento (Gutjahr, 1999). Este resultado é coerente com a intuição descrita anteriormente: quando não se sabe onde estão os defeitos, a diversidade dos testes é uma boa estratégia.

Foram propostas diferentes maneiras (ou perspectivas) para se analisar a diversidade do teste. Cada maneira leva em consideração um tipo de informação distinto, por exemplo: i) a diversidade com que os possíveis comportamentos do software – descritos em um modelo – são exercitados; ii) a diversidade com que a implementação do software é exercitada durante a execução do software em teste; iii) a diversidade de saídas produzidas pelo software como resultado; ou, iv) a diversidade de alocação de dados no domínio de entrada do software (DOTG-ID). Independentemente da perspectiva, é definido em um guia os aspectos a serem considerados para se instanciar uma perspectiva do DOTG; este guia é referido como “meta-modelo de diversidade”.

Foi desenvolvida a perspectiva do domínio de entrada para a diversidade (DOTG-ID). Esta variação da DOTG considera a posição dos dados de teste no domínio multidimensional de entrada do software para calcular a diversidade. A posição de cada dado de teste neste domínio é definida pelos valores de suas variáveis de entrada. Foram propostas: uma medida de distância entre dados de teste e uma medida de diversidade de conjuntos de teste. Foi descrito como é feito o mapeamento entre o domínio de busca das meta-heurísticas e o domínio de entrada do software em teste para a conversão de: tipos de variáveis, estruturas de dados, ou valores de entrada.

A DOTG-ID foi analisada conceitualmente com a ajuda de modelos que descrevem como os defeitos existentes podem (ou não) resultar em falhas do software, e de modelos que analisam como são formadas as partes do domínio de entrada associadas aos dados de teste que revelam os defeitos: as regiões de falha. Embora, de forma geral, não seja possível

garantir comportamento contínuo em sistemas de software (Hamlet, 2002), as regiões de falha tendem a ser contínuas (ou compactas) (Ammann e Knight, 1988), (Bishop, 1993), (Dunham e Finelli, 1990), (Schneckenburger e Mayer, 2007). A estratégia da DOTG-ID de cobrir uniformemente o domínio de entrada do software e exercitar os limites desse domínio estabelece um crivo para identificar (filtrar) regiões de falha.

A geração automática de conjuntos de teste de alta diversidade (DOTS) é feita por meio de meta-heurísticas. Foram desenvolvidas três meta-heurísticas: a SA-DOTG, baseada em Recozimento Simulado; a GA-DOTG, baseada em Algoritmo Genético; e a SR-DOTG, baseada na dinâmica de sistemas de partículas.

As meta-heurísticas SA-DOTG e GA-DOTG são ferramentas genéricas, potencialmente aplicáveis para a geração de conjuntos DOTS para qualquer perspectiva (também para a perspectiva do domínio de entrada – DOTG-ID).

A meta-heurística Repulsão Simulada (SR-DOTG) é específica para a perspectiva do domínio de entrada – DOTG-ID. É baseada na dinâmica de sistemas de partículas. Partículas (dados de teste) eletricamente carregadas resultam em forças eletrostáticas resultantes e consequentes movimentos dos dados de teste no domínio de entrada. Ao longo das iterações, os movimentos diminuem e o conjunto de dados tende a se estabilizar em uma situação de alta diversidade.

Foram realizadas avaliações empíricas. Em uma primeira etapa, as meta-heurísticas foram utilizadas para gerar conjuntos DOTS. Devido à superioridade notada, a SR-DOTG foi escolhida para as avaliações seguintes: uma simulação, e a aplicação para o teste de programas selecionados.

A simulação considerou um domínio de entrada bidimensional com variáveis do tipo real. Regiões de falha dos tipos bloco, faixa e ponto, com diferentes valores de taxa de falhas, foram aleatoriamente criadas neste domínio. Para cada taxa de falha, foram considerados vários tamanhos de conjuntos de teste. A proporção de conjuntos DOTS que encontram a região de falha (de um total de 2000 conjuntos) foi comparada à proporção

esperada para os conjuntos RTS. Os resultados mostram uma maior eficácia dos DOTS, em especial para regiões de falha compactas.

Em seguida, foi avaliado o efeito da diversidade nos valores de cobertura alcançados em programas. Conjuntos RTS e conjuntos DOTS foram executados em programas e os valores de cobertura de fluxos de dados e o escore de mutação foram avaliados para vários tamanhos de conjuntos de teste. Os resultados, significativos estatisticamente, indicam que, na maioria das situações, os conjuntos DOTS atingem valores de cobertura maiores que os alcançados pelos conjuntos RTS de mesmo tamanho.

6.2. Contribuições

Principais Contribuições

C1. A utilização de meta-heurísticas para a geração de dados de teste que satisfaçam uma propriedade alvo visando a avaliação de funcionalidade.

As abordagens SBST presentes na literatura têm evoluído com uma grande ênfase em exercitar elementos requeridos por técnicas e critérios já conhecidos (elementos estruturais como ramos e caminhos, elementos de modelos como transições e sequências de estados) ou avaliar propriedades não funcionais (ex: tempo de resposta). Este trabalho distingue-se dos outros em SBST por ter como alvo a criação de conjuntos de teste que apresentam uma propriedade (a diversidade) que pode ser caracterizada de forma independente da cobertura de elementos requeridos por técnicas e critérios conhecidos.

Pode-se considerar que este trabalho representa um passo em direção à ideia de explorar o potencial das meta-heurísticas como ferramentas para o desenvolvimento de novas técnicas e critérios de teste.

C2. A proposta de utilizar a propriedade de diversidade de conjuntos de teste como uma técnica de geração de dados de teste.

O conceito geral de diversidade (“multiplicidade de diferenças”) foi a base inicial para análise de aspectos que tornam um dado de teste diferente dos outros. Conceitos e

ideias que influenciaram esta análise foram: o conceito de continuidade em software; modelos de confiabilidade de software; e modelos de teste baseado em particionamento. Embora não tenha influenciado de forma decisiva no desenvolvimento da técnica proposta, o comportamento do sistema imune, descrito no Apêndice C, também motivou este trabalho.

C3. A proposta de diferentes perspectivas para a diversidade e de um meta-modelo para direcionar a instanciação dessas perspectivas.

Cada perspectiva focaliza um tipo de informação a ser considerada para medir a diversidade de conjuntos de teste (e para gerá-los). O meta-modelo fornece um guia com aspectos genéricos a serem considerados para instanciar a geração de dados de teste orientada à diversidade para uma perspectiva específica.

C4. O desenvolvimento e a avaliação da perspectiva do domínio de entrada para o teste orientado à diversidade (DOTG-ID).

Definição dos conceitos de distância entre dados de teste e de diversidade do conjunto de teste. Análise da *DOTG-ID* em relação a modelos de detecção de defeitos. Avaliação empírica da *DOTG-ID* por meio de uma simulação e por meio de avaliação de cobertura de fluxo de dados e de escore de mutação.

C5. A concepção, o projeto e a implementação da meta-heurística Repulsão Simulada para a geração de dados de teste (SR-DOTG).

A simulação de repulsão entre os dados de teste faz com que eles se posicionem no domínio de entrada do programa de forma a gerar conjuntos de teste de alta diversidade. A SR-DOTG apresentou um desempenho expressivamente superior às meta-heurísticas baseadas em Recozimento Simulado e em Algoritmo Genético.

Contribuições Secundárias

C6. O projeto e a implementação de meta-heurísticas para a geração de conjuntos de dados de teste de alta diversidade (DOTS).

Foram desenvolvidas a SA-DOTG, baseada em Recozimento Simulado e a GA-DOTG, baseada em Algoritmo Genético. Ambas utilizam o valor de diversidade dos conjuntos de teste para direcionar a busca por conjuntos DOTS. Estas meta-heurísticas são potencialmente aplicáveis para a geração de conjuntos DOTS para qualquer perspectiva (também para a perspectiva do domínio de entrada – DOTG-ID).

C7. A Identificação de pontos chave para definição de novas abordagens SBST e de um mecanismo genérico para a geração de dados baseada em SBST.

A avaliação de vários trabalhos que envolvem a geração de dados de teste baseada em busca (SBST) permitiu identificar questões genéricas a serem tratadas para se desenvolver abordagens SBST. Esta avaliação permitiu também propor um mecanismo que captura em linhas gerais o funcionamento das meta-heurísticas no processo de geração de dados. O Capítulo 3 apresenta este conteúdo:

- A definição de um guia de sete passos que descrevem os pontos chave para o desenvolvimento de novas abordagens SBST.
- A definição de um mecanismo genérico e simples para a geração de dados baseada em SBST composto por seis passos de processamento.

6.3. Trabalhos Futuros

Principais Trabalhos Futuros

TF1. Investigar outras propriedades, ou combinação de propriedades, como alvo para a geração de dados de teste visando à avaliação de funcionalidade.

Esta linha de pesquisa consiste em investigar outras propriedades de conjuntos de teste, que não a diversidade, que possam ser úteis tendo em vista o objetivo do teste de software de revelar defeitos e de contribuir para estabelecer a confiança na adequação do produto testado. Por exemplo, podem ser levados em conta para a geração de dados aspectos como o custo do teste, os riscos do software, a complexidade do software, etc.

TF2. Investigar outras medidas de distância ou dissimilaridade, que não a distância euclidiana, para calcular distâncias entre dados de teste e diversidade de conjuntos de teste na abordagem DOTG-ID (perspectiva domínio entrada).

Esta linha de pesquisa deve considerar medidas alternativas de distância entre dados numéricos, como a Distância de Manhattan, ou a Distância de Cosseno. Caso as variáveis de entrada sejam representadas com valores binários, medidas como a Distância de Hamming, ou a Distância de Jaccard podem ser consideradas. Mais genericamente, a distância entre strings de caracteres quaisquer pode ser avaliada por meio de medidas como a Distância de Levenshtein (Distância de Edição), ou a distância de Smith-Waterman. Existem muitos trabalhos que abordam medidas de distância e de similaridade em várias áreas, exemplos: (Santini e Jain, 1999) e (Strehl et. al, 2000).

No caso de variáveis estabelecerem um conjunto de valores possíveis (ou categorias) não contínuos o conceito de distância possivelmente será mais complexo. Exemplos: estabelecer distâncias entre elementos: vermelho, verde, azul, laranja; ou entre os elementos: triângulo, quadrado, pentágono, círculo (Boriah et. al, 2008)

Domínios específicos também podem ser considerados; por exemplo: diversidade de programas para testar compiladores, diversidade de imagens para testar software de processamento de imagens, etc.

Cada medida distinta das descritas acima pode ter a sua adequação à geração de dados avaliada preliminarmente e, eventualmente, gerar uma nova abordagem DOTG-ID. Esta linha deve considerar medidas utilizadas em outras áreas como: ecologia e diversidade genética; além de áreas que requerem medições de distância e dissimilaridade como: clusterização de dados e recuperação de informação.

TF3. Desenvolver e validar outras perspectivas para a diversidade.

Esta linha de pesquisa pode ter como ponto de partida o meta-modelo para diversidade proposto. Devem ser consideradas as possíveis informações relacionadas a uma perspectiva e como essas informações serão utilizadas para gerar conjuntos de teste de alta diversidade.

Por exemplo, para a perspectiva que leva em conta informações sobre a execução do software durante o teste (*DOTG-EI*, *EI: Execution Information*), podem ser considerados: caminhos executados; associações de fluxo de dados exercitadas (ADU); mutantes exercitados; estados de dados criados durante a execução. Estas informações podem ser estruturadas em diferentes níveis (ou granularidades) para o cálculo da diversidade de conjuntos de teste.

O Apêndice D fornece um esboço da perspectiva que leva em conta informações sobre a execução do software durante o teste (*DOTG-EI*, *EI: Execution Information*).

Trabalhos Futuros Secundários

TF4. Desenvolver e avaliar um modelo que considere múltiplas perspectivas para a diversidade.

A partir do resultado da avaliação de cada perspectiva para a diversidade (ex: DOTG-ID, DOTG-IE delineada no Apêndice D e outras), desenvolver um modelo que considere as múltiplas perspectivas, buscando a intersecção de características e informações, ou a união de características e informações.

TF5. Avaliar a aplicação de outras meta-heurísticas para a técnica DOTG.

Meta-heurísticas e suas características foram descritas no Capítulo 3. Independentemente da perspectiva da diversidade, alternativas às utilizadas neste trabalho devem ser avaliadas. Por tratarem a diversidade de forma inerente em seus mecanismos de busca, duas alternativas se destacam: a meta-heurística *CLONALG – CLONal selection ALGORITHM* (De Castro, 2001), (De Castro e Von Zuben, 2002) e a Busca Dispersa – *Scatter Search* (Glover, Laguna e Martí, 2000).

TF6. Avaliar a utilização de conjuntos de teste de alta diversidade para a estimação de confiabilidade de software.

Como descrito no Capítulo 4 e mais detalhadamente no Apêndice C, modelos de confiabilidade que consideram cobertura do código e número de dados de teste executados tendem a ser mais precisos nas estimativas do que os que consideram o tempo de execução do software. Esta linha de pesquisa consiste em avaliar a adequação (ou não) da diversidade de conjuntos de teste como informação para estimar a confiabilidade com o uso de modelos de confiabilidade.

Outra linha de trabalho relacionada à confiabilidade seria estabelecer uma função de diversidade que, ao ser maximizada, implique em: (a) maximizar a cobertura de elementos requeridos (ex: Associações Definição-Uso); e (b) aproximar a alocação de dados o tanto quanto possível do Perfil Operacional do software em teste. Este conjunto de teste resultante teria potencialmente as propriedades de: atingir boa capacidade de revelar defeitos (devido ao objetivo a) e servir como base para a estimativa de confiabilidade do software em uso (devido ao objetivo b).

TF7. Combinar ideias da técnica DOTG com a Seleção de Dados de Teste por Análise e Clusterização de Perfis de Execução (Leon e Podgurski, 2003).

A abordagem de Leon e Podgurski (2003) considera como os perfis de execução induzidos pelos casos de teste estão distribuídos em um espaço de perfis. Uma filtragem de clusters automaticamente particiona o conjunto de casos de teste de acordo com a similaridade dos perfis, avaliada por meio de uma medida de dissimilaridade. Testes são então selecionados de cada um dos clusters para formar o conjunto de casos de teste.

Nesta linha de pesquisa pode ser avaliada a utilização de conjuntos de alta diversidade para executar o software e gerar os perfis de execução. A hipótese é que conjuntos DOTS vão ter maior capacidade de induzir diferentes perfis em comparação com a geração aleatória de dados.

Outra linha consiste em empregar uma meta-heurística para gerar conjuntos DOTS; por exemplo, com a técnica DOTG-IE (vide Apêndice D) e aplicar algoritmos de

clusterização para analisar os perfis de execução explorados na busca e eventualmente retro-alimentar a busca.

TF8. Desenvolver uma abordagem incremental para o teste orientado à diversidade.

Esta linha consiste em desenvolver uma abordagem que permita gerar um conjunto de testes com alta diversidade, mas considerando um conjunto de dados de teste pré-existente. Isto é, dado um conjunto de teste $T1$ com n dados de teste, gerar um conjunto $T2$ que preserve os n dados de teste de $T1$ e adicione m dados de teste de maneira a maximizar a diversidade do conjunto resultante $T3$ ($T3 = T1 \cup T2$). $T3$, por sua vez, poderia ser a base para o próximo incremento. Esta linha pode ser desenvolvida para cada perspectiva de diversidade.

TF9. Avaliar a adequação do uso de outros mecanismos do sistema imune para aprimorar a geração de dados de teste.

No Apêndice C são descritos os mecanismos usados pelo sistema Imune para proteger seres vivos de entidades perigosas. Os mecanismos de promoção da diversidade e a estratégia de cobertura do espaço de possíveis patógenos com anticorpos guardam semelhanças com a técnica DOTG-ID. Esta linha de pesquisa consiste em avaliar se outros mecanismos do sistema imune, como a memória e capacidade de generalização, podem inspirar novas técnicas de teste.

TF10. Avaliar a utilização da meta-heurística Repulsão Simulada (SR-DOTG) com outros propósitos que não a geração de dados de teste.

Esta linha trata da avaliação da adequação do uso da ideia da Repulsão Simulada em outros problemas, por exemplo, para iniciar pesos de redes neurais (De Castro e Von Zuben, 2001c) ou populações de Algoritmos Genéticos.

Pode também ser avaliado o uso de repulsão simulada como recurso para preservar a diversidade durante a busca com meta-heurísticas baseadas em populações.

Apêndice A – Técnicas de Teste:

Definições e Detalhamentos

Teste Combinatório: Definições e Notação

Um software em teste (SUT) recebe como entrada n parâmetros c_i ($i = 1, 2, \dots, n$) que podem influenciar a execução do SUT. Um parâmetro c_i pode possuir a_i valores discretos de um conjunto finito V_i , $a_i = |V_i|$. Assume-se que os parâmetros sejam independentes, ou seja, nenhum deles é determinado pelos outros. Maiores detalhes sobre os conceitos apresentados nesta seção, vide (Nie e Leung, 2009).

Um conjunto de relações de interação R registra todas as interações existentes entre os parâmetros, por exemplo, $R = \{\{c_1\}, \{c_2, c_3\}\}$. O elemento $\{c_1\}$ em R significa que todos os parâmetros para c_i em V_i podem afetar o SUT e podem provocar uma falha do software. O elemento $\{c_2, c_3\}$ em R significa que existem interações entre c_2 e c_3 , isto é, entre pares em $V_2 \times V_3$, que podem provocar uma falha do software. R pode ser determinado a partir da especificação do software e pode ser uma base para a definição de requisitos no teste combinatório, especificando quais combinações devem ser cobertas pelo teste.

Um caso de teste t é uma n -tupla (v_1, v_2, \dots, v_n) onde $v_1 \in V_1, v_2 \in V_2, \dots, v_n \in V_n$. t -all denota todos os possíveis casos de teste para o SUT.

Quando valores específicos para os parâmetros são combinados ocorre uma interação entre parâmetros que podem provocar uma falha. Um *Esquema k -valores* ($k > 0$) para um SUT é uma n -tupla $(—, v_{n_1}, \dots, v_{n_k}, \dots)$ na qual k parâmetros possuem valores fixos (definidos) e os demais parâmetros podem assumir valores permitidos para eles (não definidos), representados como um “—” (significando “não importa”). Quando $k = n$, a n -tupla é um caso de teste para o SUT. A falha de um SUT pode ser provocada pelo valor de um parâmetro, ou seja, por algum *Esquema 1-valor*; pode ser provocada por uma interação

de algum par de parâmetros, ou seja, por algum *Esquema 2-valores*, e assim por diante (*Esquemas k-valores*, $k > 2$). Um conceito análogo é definido por Kuhn, o Número de Interações de Falhas Causadoras de Defeitos (*failure-triggering fault interaction number - FTFI*)¹⁴ é o número de condições (associadas aos parâmetros) requeridas para se causar uma falha (Kuhn et al., 2004), (Kuhn et al., 2009).

Um Vetor Ortogonal (*Orthogonal Array*) denotado por $OA(N, t, n, (a_1, a_2, \dots, a_n))$ é um vetor $N \times n$ com as propriedades: cada coluna i ($1 \leq i \leq n$) contém apenas elementos do conjunto V_i ($a_i = |V_i|$); as linhas de cada sub-vetor $N \times n$ cobrem todas as t -tuplas de valores das t colunas exatamente λ vezes. t é referido como força da cobertura de interações; n é o número de parâmetros ou componentes (grau); e a_1, a_2, \dots, a_n são os números de possíveis valores para cada parâmetro (ordens).

Por requerer que cada combinação seja coberta um mesmo número de vezes (λ vezes), pode ocorrer que, dependendo dos valores de t e de a_i , o vetor ortogonal aplicável tenha $\lambda > 1$, tornando o conjunto de teste gerados relativamente grande. Uma alternativa aos vetores ortogonais que trata esta questão são os vetores de cobertura.

Um Vetor de Cobertura denotado por $CA(N, t, n, (a_1, a_2, \dots, a_n))$ é um vetor $N \times n$ com as propriedades: cada coluna i ($1 \leq i \leq n$) contém apenas elementos do conjunto V_i ($a_i = |V_i|$); as linhas de cada sub-vetor $N \times n$ cobrem todas as t -tuplas de valores das t colunas *pelo menos uma vez*. t é referido como força da cobertura de interações; n é o número de parâmetros ou componentes (grau); e a_1, a_2, \dots, a_n são os números de possíveis valores para cada parâmetro (ordens).

A distinção entre um vetor de cobertura e um vetor ortogonal é que o primeiro relaxa a restrição de que cada combinação seja coberta exatamente o mesmo número de vezes.

Permitir números de possíveis valores distintos para cada parâmetro (a_1, a_2, \dots, a_n), conforme as definições anteriores, é referido por alguns autores como *Vetor de Cobertura Mesclada de Níveis (Mixed Level Covering Array)*, ao invés de Vetor de Cobertura simplesmente. Nestes casos o vetor de cobertura é definido Cobertura como $CA(N, t, n, a)$,

¹⁴ Este conceito seria melhor nomeado como *failure-triggering parameter interaction number*.

denotando que todos os parâmetros possuem uma mesma ordem (a). Esta restrição limita a aplicação desses vetores em situações reais, em que tipicamente diferentes parâmetros possuem diferentes números de possíveis valores distintos.

Um Vetor de Força Variável (*Variable Strenght Covering Array*) é um método que utiliza vetores de cobertura que oferecem diferentes forças de cobertura (t) para diferentes grupos de parâmetros. O ponto é que as interações não existem uniformemente: alguns parâmetros podem ter fortes interações uns com os outros, enquanto outros podem ser independentes, ou fracamente relacionados. Parâmetros altamente relacionados, ou que por algum motivo precisam ser exercitados de forma rigorosa, podem ser testados usando vetores com força maior do que os outros parâmetros.

Caracterização do Teste Combinatório

Tópicos importantes na pesquisa e na aplicação do teste combinatório podem ser encontrados em (Nie e Leung, 2009). Um breve sumário é fornecido nesta seção.

A *modelagem do Software em Teste (SUT)* é uma atividade fundamental para a aplicação bem estruturada do teste combinatório. O tratamento de elementos importantes na aplicação da técnica e a eficácia do teste requerem a elaboração de um modelo preciso do software. Esta modelagem compreende quatro aspectos:

1. Parâmetros que podem afetar o SUT, que podem ser de várias naturezas, tais como parâmetros de configuração, ou entradas de usuários;
2. Valores que devem ser selecionados para cada parâmetro. Parâmetros podem ser associados a um conjunto finito valores específicos (ex: estados de um país), ou podem assumir valores contínuos (ex: temperatura, idade, etc) levando a um número infinito de valores. Nestes casos valores limite específicos, ou classes de equivalências, podem ser empregados para seleção de valores;

3. Relações de interação que existem entre os parâmetros. Identificar a possível existência de: parâmetros que não interagem com nenhum outro parâmetro; parâmetros que possuem forte interação com outros.
4. Restrições que existem entre valores dos diferentes parâmetros. Identificar se valores específicos de um parâmetro conflitam com valores de outros parâmetros.

Feita esta modelagem, as estratégias de teste combinatório envolvem o tratamento dos seguintes elementos: seleção de uma estratégia de combinação; definição de casos de teste semente; tratamento de restrições; e aplicação de algum algoritmo para a seleção de casos de teste. Estes elementos são descritos a seguir.

Seleção de uma estratégia de combinação: consiste em especificar o tipo de cobertura a ser utilizada no teste, por exemplo: Um Fator Por Vez, Vetor Ortogonal, ou Vetor de Cobertura.

Definição de sementes: casos de teste a serem incluídos no conjunto de casos de teste sem nenhuma modificação. Permitem especificar explicitamente combinações de valores a serem usados no teste, por exemplo, situações críticas, ou situações muito prováveis no uso do SUT. Sementes também são úteis em uma estratégia incremental e adaptativa de Vetores de Cobertura, na qual conjuntos de casos de teste com vetores de cobertura de baixa força (t) são utilizados inicialmente, e, à medida que os recursos permitam, vetores de cobertura de forças maiores são gerados. Em cada estágio, os casos de teste gerados em estágios anteriores são reusados como sementes.

Definição de Restrições: podem ocorrer situações nas quais alguma combinação de valores para os parâmetros é inválida, levando à necessidade de tratar restrições (aspecto já citado no item 4 anterior). Ignorar restrições pode levar a geração de muitos casos de teste inválidos, com o potencial de perda de cobertura de outras combinações válidas. Alguns trabalhos abordam esta questão, por exemplo, Cohein et al., (1997) propõem uma técnica para compilar restrições, gerando expressões lógicas para testes “não permitidos”, tratadas de forma integrada aos algoritmos de geração de dados.

Técnicas para a geração de casos de teste: tais técnicas tratam do problema NP-completo da geração de Vetores de Cobertura. Diferentes linhas para tratar este problema incluem: algoritmos *greedy*; busca heurística; (ambas anteriores referidas como abordagens computacionais); e métodos matemáticos.

Algoritmos *greedy* constroem um conjunto de casos de teste de forma que cada novo caso de teste adicionado ao conjunto cobre o maior número possível de combinações ainda não cobertas. Essas técnicas podem ser do tipo “Uma-linha-por-vez” (*One-row-at-a-time*), ou do tipo “Na-ordem-dos-parâmetros” (*In-parameter-order – IPO*). Algoritmos do primeiro tipo constroem uma linha do vetor por vez, até que todas as combinações requeridas tenham sido cobertas, um exemplo é o AETG (Cohen D. et al., 1997) (combinações requeridas são referidas como *t-conjuntos*, sendo *t* a força da cobertura). Algoritmos do segundo tipo iniciam a geração de dados criando todos os *t-conjuntos* dos primeiros *t* fatores. Os algoritmos então expandem incrementalmente este conjunto, tanto na horizontal (novas linhas) quanto na vertical (novos parâmetros). Lei et al. descrevem um algoritmo deste tipo (Kacker et al., 2008).

Técnicas de busca aplicadas à geração de vetores de cobertura incluem: Subida de Encosta; Busca Tabu; Recozimento Simulado; Algoritmos Genéticos e Colônia de Formigas. Essas técnicas iniciam por um conjunto de teste pré-existente e aplicam transformações nos conjuntos de teste até que ele cubra todas as combinações. Por exemplo, Ghazi usam um Algoritmo Genético para gerar conjuntos de teste que maximizam a cobertura de pares dado um número pré-definido de casos de teste (Ghazi e Ahmed, 2003). Em geral, técnicas de busca heurística produzem conjuntos de dados de teste menores do que os produzidos usando algoritmos *greedy*, mas requerem um maior custo de computação.

Técnicas matemáticas para computar vetores de cobertura são normalmente extensões de métodos matemáticos para construção de vetores ortogonais. Como exemplo tem-se a técnica apresentada por Williams (Williams e Probert, 2002) que constrói recursivamente conjuntos de teste maiores a partir da concatenação e adaptação de conjuntos menores. Técnicas matemáticas não são genéricas e impõem restrições em relação a aspectos do

domínio em que serão aplicadas (ex: números de parâmetros e número de valores para os parâmetros). Por outro lado, tais técnicas tendem a ser muito rápidas e, em condições especiais, gerar conjuntos mínimos de teste.

Processo de Teste no Teste Combinatório

Krishnan et. al, definem uma sequência de passos para aplicar um método de teste combinatório (especificamente a *OATS – Orthogonal Array Based Testing Strategy*) em aplicativos de telefonia móvel (Krishnan et. al, 2007). Embora definido em um contexto específico, o processo proposto é genérico o suficiente para ser útil em outros domínios.

Passo 1: particionar os requisitos em grupos lógicos. Em geral diferentes funções são controladas por variáveis distintas, portanto é necessário identificar as funções e as variáveis que controlam cada função.

Passo 2: definir o escopo e a abordagem do teste. Diferentes grupos de requisitos podem ser testados com diferentes níveis de rigor, por exemplo, testar a interação de pares de valores para as variáveis, ou testar apenas valores simples (classes de equivalência). A abordagem depende também do nível do teste (unidade, sistema, etc.).

Passo 3: identificar, organizar e validar as variáveis. Coletar os requisitos a serem testados e identificar as variáveis para o teste (geralmente nomes que aparecem nos requisitos). Classificá-las como “passivas” quando elas não afetam os requisitos a serem testados, independentemente dos seus valores; ou “ativas”, quando podem assumir múltiplos valores e influenciam na funcionalidade testada. As variáveis devem ser validadas para identificar se elas são mutuamente independentes ou não.

Passo 4: associar níveis às variáveis. Para cada variável identificar os seus níveis (possíveis valores distintos). Particionamento de equivalência e análise de valores limites podem ser usados para determinar os níveis.

Passo 5: identificar restrições. Quando os níveis são associados às variáveis é possível que eles restrinjam os níveis de outras variáveis. Exemplo: quando, para evitar a

geração de testes inválidos, pares de variáveis devem ser analisadas visando a identificar restrições.

Passo 6: gerar os testes. Utilizar a ferramenta de geração de dados para gerar os testes de acordo com as informações levantadas nos passos anteriores.

Passo 7: otimizar os testes. Identificar testes inválidos, que podem existir por erros em etapas anteriores. Se o número de testes for maior que o esperado, avalie a variável com mais níveis, reduzindo-os caso possível. Adicionar combinações de níveis mais altos para situações críticas específicas, não contempladas pelos testes gerados.

Passo 8: adicionar observações e resultados esperados. Para obter casos de teste completos é necessário identificar observações a serem feitas durante a execução e os resultados esperados. É importante também checar a rastreabilidade dos casos de teste para os requisitos para identificar testes inválidos, ausências na especificação de requisitos, ou omissões nos testes.

Teste Baseado em Modelos

O Teste de Software Baseado em Modelos (*Model-Based Software Testing*) vem ganhando espaço nos últimos anos devido ao avanço o paradigma de orientação a objetos e à evolução da adoção de modelos na engenharia de software – o desenvolvimento de software baseado em modelos (Stahl e Völter, 2006).

El-Far e Whittaker definem um modelo como “uma descrição do comportamento de um software” (El-Far e Whittaker, 2001). Bertolino et al. definem o Teste Baseado em Modelos (TBM) como “a derivação de casos de teste de um modelo representado o comportamento do software” (Bertolino et al., 2005). Dalal et al. enfatizam o uso de um modelo de dados que especifica as entradas recebidas pelo software (interface do software) (Dalal et al., 1999). O teste baseado em modelos refere-se, portanto, a abordagens que baseiam as tarefas de teste, como a geração de dados de teste e a avaliação de resultados, em um modelo do software em teste.

Caracterização do Teste Baseado em Modelos

Uma taxonomia para o teste baseado em modelos é proposta por Utting et al. (Utting e Legeard, 2006), (Utting et al., 2006). São identificadas sete diferentes dimensões para analisar estas técnicas e são discutidas possíveis instanciações em cada dimensão. Essas dimensões, embora ortogonais, influenciam-se mutuamente e relacionam-se essencialmente aos aspectos: modelos, geração de teste e execução de teste. A seguir são sumariamente descritas as dimensões.

Sujeito do modelo (*model subject*): o modelo pode referir-se ao software em teste ou ao ambiente do teste. No primeiro caso o modelo pode servir como oráculo ou como estrutura base para a geração dos testes. No segundo caso o modelo restringe as possíveis entradas a serem utilizadas.

Nível de redundância do modelo (*model redundancy level*) refere-se à distinção entre dois cenários possíveis: o uso de um modelo para a geração automática de código e de casos de teste, ou o uso de um modelo específico para o teste, criado a partir da especificação.

Características do modelo (*model characteristics*) relacionam-se ao não determinismo, à incorporação de aspectos temporais, ou à natureza (contínua ou baseada em eventos) do sistema. Este aspecto é relevante, por exemplo, no teste de sistemas de tempo real.

Paradigma do modelo (*model paradigm*) diz respeito à notação utilizada para modelar o comportamento do sistema. Foram caracterizados os seguintes paradigmas. *Notações baseadas em estados* modelam o sistema como uma coleção de variáveis, que representam o estado interno do sistema. Operações que modificam as variáveis são definidas usando pré-condições e pós-condições (exemplos: Z, B, VDM, JML). *Notações baseadas em transição* focam na descrição de transições entre diferentes estados do sistema, representadas tipicamente por meio de nós e arcos, como em máquinas de estados finitos. Esses modelos podem ser adicionados de variáveis, hierarquia, ou paralelismo, para torná-los mais expressivos (exemplos: máquinas de estados UML, *statecharts*, *simulink*

stateflow). *Notações baseadas em história* modelam o sistema em termos de possíveis comportamentos em relação ao tempo. Podem ser discretas ou contínuas e utilizar diferentes lógicas temporais. *Notações funcionais* descrevem o sistema como uma coleção de funções matemáticas (exemplos: funções de primeira ordem, notação HOL). *Notações operacionais* descrevem o sistema como uma coleção de processos executáveis em paralelo. Aplicam-se à descrição de sistemas distribuídos e de protocolos de comunicação (exemplos: CSP, CCS, Redes de Petri). *Notações estocásticas* descrevem o sistema por meio de um modelo probabilístico de eventos e valores de entrada (exemplo: Cadeias de Markov). *Notações de fluxo de dados* concentram-se nos dados (exemplos: Lustre, diagramas de blocos).

Critério de seleção de teste (*test selection criteria*) refere-se aos tipos de critérios de seleção de dados de teste, agrupados como se segue. *Critérios de cobertura estrutural do modelo* exploram a estrutura do modelo, como nós, arcos e transições em modelos baseados em transições, ou condições em notações que envolvem pré e pós-condições. *Critérios de cobertura de dados* tratam da escolha de valores para o teste, usando conceitos como classes de equivalência, valores limite e análise de domínio. *Critérios de cobertura baseados em requisitos* associam explicitamente os elementos do modelo com os requisitos do sistema em teste. *Especificação ad-hoc de casos de teste* consiste da especificação direta de casos de teste, eventualmente determinado alguns componentes críticos de um modelo a serem exercitados. *Critérios aleatórios* determinam um padrão de uso do sistema em teste, modelando a probabilidade de ações segundo este padrão. *Critérios baseado em defeitos* envolvem a geração de testes direcionados a tipos de defeitos no sistema, como exemplo o critério de cobertura de mutantes.

Tecnologia de geração de teste faz referência à automação da geração dos casos de teste. Dado um modelo do sistema em teste e uma especificação abstrata de um caso de teste (como na forma de elementos requeridos a serem exercitados), os casos de teste podem ser gerados usando as seguintes técnicas. Geração aleatória pela amostragem do espaço de entrada, definindo valores aleatórios, seqüências de eventos aleatórias, ou caminhos aleatórios (*random walks*) no modelo. Algoritmos de busca em grafo, como

carteiro chinês, podem ser aplicados para exercitar arcos em modelos. Checagem de modelos (*model checking*) pode ser empregada para gerar contra exemplos para propriedades. Execução simbólica, representando classes de execuções, baseadas no processamento simbólico das especificações, pode ser empregada para instanciar valores concretos para os casos de teste. Prova dedutiva de teoremas também pode ser usada para gerar casos de teste.

Geração de testes on-line ou off-line diz respeito ao tempo da geração e da execução dos dados de teste. No caso da geração on-line os algoritmos podem reagir às saídas realmente produzidas pelo software em teste, ou a eventos produzidos na execução. A execução do dado de teste é monitorada, por exemplo, para avaliar o caminho executado no software. No caso da geração off-line os dados de teste são gerados em uma etapa anterior à execução. Nestes casos os dados de teste podem ser gerados uma vez e executados várias vezes; a geração e execução podem ser feitas em momentos e máquinas diferentes; é também possível processar o conjunto de testes (ex: minimizá-lo) antes da execução.

Em uma revisão sistemática as abordagens de TBM (relatadas em 202 artigos) foram classificadas nas seguintes categorias (Dias Neto et al., 2007):

- i. Modelos representando informações sobre os requisitos e descritos usando diagramas UML;
- ii. Modelos representando informações sobre os requisitos e descritos usando qualquer notação não UML;
- iii. Modelos representando informações da estrutura do software (arquitetura, componentes, interfaces) descrita usando diagramas UML; e,
- iv. Modelos representando informações da estrutura do software usando qualquer notação não UML.

O número de trabalhos selecionados foi de 26, 93, 21 e 62 para categorias i a iv, nesta ordem. Isto indica que UML tem papel importante nas abordagens de TBM, no entanto “outros modelos” considerados ao todo são mais empregados.

Esta revisão apresenta informações que caracterizam os trabalhos em TBM:

- Abordagens focam principalmente o nível teste de sistema (66% dos trabalhos), seguido do teste de integração (22%), teste de unidades (10%), e de regressão (5%).
- Os modelos mais usados são: *statecharts* UML (27 trabalhos), diagramas de classe UML (19), diagramas de sequência UML (19) e máquinas de estados finitos (7).
- O custo e a complexidade da aplicação do TBM relacionam-se aos fatores: a complexidade do modelo; a existência de ferramentas de apoio; e o nível de automação das atividades de teste. Foram identificadas classes de custo do TBM e características relacionadas às classes. *Custo alto*: atividades manuais, ausência de ferramentas de apoio, tradução entre modelos requerida. *Custo intermediário*: modelos com notação complexa e uso de ferramentas proprietárias. *Custo baixo*: uso de modelos conhecidos (UML, MEF), alto nível de automação, ferramentas de apoio disponíveis.

Shafique e Labiche apresentam uma revisão sistemática sobre ferramentas de suporte para o TBM (Shafique e Labiche, 2010). O trabalho tem como escopo ferramentas “baseadas em estados” (*state-based model based testing tools*), tanto comerciais quanto voltadas à pesquisa. A questão investigada é quais são as ferramentas de TBM e quais são as suas características. É identificado o nível de suporte automatizado para atividades como: criação do modelo; verificação do modelo; depuração; rastreabilidade de requisitos; e criação de *drivers*, *stubs* e oráculos. Outro aspecto analisado, e que é particularmente interessante no contexto deste trabalho, é relacionado a critérios de teste – suporte à geração de casos de teste e à avaliação de adequação de casos de teste.

Foram identificados quatro grupos de critérios: baseado em modelos de fluxo (*model-flow criteria*); baseado em fluxos de scripts (*script-flow criteria*); baseado em dados (*data criteria*); e baseado em requisitos (*requirement criterion*). Esses grupos são descritos brevemente a seguir.

Critérios baseado em modelos de fluxo (*model-flow criteria*) definem elementos do modelo de estados a serem exercitados, tais como a cobertura de estados, de transições, de

pares de transições, de transições paralelas, ou de cenários. Nos critérios baseados em fluxos de scripts (*script-flow criteria*) uma linguagem de scripts, ou pré-condições e pós-condições, permitem especificar o comportamento do software em outros aspectos além de estados. Esses critérios requerem que partes da especificação sejam exercitadas: cobertura de comandos, cobertura de ramos, cobertura de condições, e cobertura de caminhos. Critérios baseados em dados (*data criteria*) referem-se à seleção de valores de entrada para a criação de casos de teste concretos a partir de casos de teste abstratos. Exemplos de critérios: um valor (selecionar apenas um valor para o teste); todos os valores; valores limites; combinação de pares de valores (*parwise*). Critérios baseados em requisitos (*requirement criterion*) baseiam-se na rastreabilidade entre requisitos e elementos do modelo.

Tarefas Fundamentais do Teste Baseado em Modelos

Segundo Dalal et al. aplicação do TBM requer: um *modelo* usado para descrever o comportamento do software; um *algoritmo* – ou *critério* – para gerar os casos de teste; e *ferramentas* que servem como uma estrutura básica para o teste (Dalal et al., 1999).

El-Far e Whittaker identificam as seguintes *tarefas fundamentais* para a realização de um teste baseado em modelos (El-Far e Whittaker, 2001):

Compreender o sistema em teste: desenvolver a compreensão sobre o software e sobre o seu ambiente. Algumas diretrizes: identificar componentes a serem testados; explorar os últimos releases do software a ser testado; coletar documentos existentes sobre o software; estabelecer comunicação com outras equipes (ex: desenvolvimento) e identificar modelos usados por elas; identificar usuários do sistema e entradas e saídas por eles produzidas para o sistema; identificar condições que restringem entradas do usuário, ou respostas do sistema; identificar a estrutura e a semântica de dados externos e internos;

Escolher um modelo: diferentes domínios de aplicação associam-se a diferentes tipos de modelos mais adequados. Exemplos em (El-Far e Whittaker, 2001): um *modelo de gramática* é uma abordagem mais direta para gerar arquivos HTML como entrada para

testar um navegador, ou para gerar expressões matemáticas para testar uma calculadora científica; Sistemas de telefonia com muitos estados e comportamentos específicos deles dependentes podem ser idealmente representados por *máquinas de estados finitos* (MEF); se o sistema possui poucos estados, mas com transições determinadas por muitas condições os “*statecharts*” podem ser adequados; se o teste deve avaliar o comportamento do software submetido a combinações de valores para as entradas, um modelo de *tabela de decisões* pode ser usado. Aspectos relacionados a habilidades e experiências da equipe e às políticas organizacionais também são importantes na escolha de modelos de teste. Dias Neto et al. destacam que o TBM e o desenvolvimento do software devem estar integrados (Dias Neto et. al, 2007). Usar modelos diferentes para o projeto do software e para o teste pode representar um esforço adicional evitável.

Notar que esta perspectiva de modelos para o teste faz com que o conceito TBM represente um espectro amplo de técnicas. Por exemplo, o teste usando tabelas de decisão, ou grafos causa-efeito (já abordados), também pode ser considerado baseado em modelos.

Construir o modelo: esta etapa consiste em aplicar um procedimento específico para gerar o modelo a ser utilizado no teste a partir das informações (documentos) originadas em etapas anteriores (análise de requisitos, projeto e codificação do software). Diretrizes para construir modelos são específicas para o modelo escolhido e podem ser encontradas na literatura, ex: (Beizer, 1990), (Binder, 2000), (El-Far e Whittaker, 2001), (Utting e Legeard, 2006), (Walton e Poore, 2000).

Gerar os dados de teste: utilizando critérios de teste associados ao modelo, selecionar os dados de testes. Por exemplo, no caso do modelo ser uma MEF podem ser utilizados critérios para a geração de seqüências de estados (caminhos na MEF) e respectivas seqüências de eventos de entrada.

Executar os testes e avaliar resultados: tipicamente os testes são executados após a geração de um conjunto de teste segundo algum critério. *Scripts* de teste são preparados para aplicar os dados de teste no software. Os resultados obtidos da execução devem ser avaliados, o que requer que as saídas produzidas e o comportamento do software sejam comparados com os esperados.

Sumário de Aplicações do Teste Baseado em Modelos

Apfelbaum e Doyle apresentam uma aplicação de modelos de comportamento no teste de integração e de sistemas (Apfelbaum e Doyle, 1997). *Scripts* de teste são definidos como seqüências de ações formadas por primitivas, que podem: prover um estímulo para o sistema, verificar uma resposta do sistema, definir parâmetros do ambiente de teste e registrar resultados. Os *scripts* de teste são definidos para cobrir caminhos no modelo de comportamento. A abordagem é aplicada em dois sistemas: um sistema de serviços de chamada telefônica de longas distâncias e um sistema de gerenciamento e rastreamento de ordens de serviço.

Clarke relata a aplicação de um modelo de comportamento para o teste de um equipamento de comutação em telefonia digital (Clarke, 1988). Os requisitos do sistema, expressos em um documento de especificação, são utilizados para a criação de um modelo de comportamento. O modelo combina conceitos de fluxo de controle, máquinas de estados finitos e máquinas de estados finitos estendida (MEFE), que associa predicados aos estados para evitar explosão de estados. Este método foi aplicado no teste de duas funcionalidades: um serviço de gerenciamento de chamadas e um serviço de portabilidade de números telefônicos.

Offutt e Abdurazik apresentam uma técnica que usa especificação UML *statecharts* para a geração de dados de teste (Offutt e Abdurazik, 1999). Os critérios de teste definidos são: *cobertura de transições*, que requer que cada transição do *statechart* seja executada, ou seja, cada precondição associada a cada estado é exercitada; *cobertura de predicados*, que requer que cada cláusula (expressão lógica) de cada predicado seja exercitada, o que leva ao teste mais rigoroso no caso de predicado compostos (com mais de uma cláusula); *cobertura de pares de transições*, que requer que cada par de transições adjacentes seja exercitado; e *seqüências completas*, que requer que toda seqüência de estados do *statechart*, que caracteriza um uso do software, seja exercitada. O *statechat* é representado como uma tabela de que estabelece predicados associados às transições e serve como entrada para uma ferramenta de geração automática de seqüências para o teste.

Kansomkeat e Rivepiboon apresentam uma abordagem semelhante, transformando *statecharts* em um diagrama intermediário, chamado de grafo de fluxo de teste, usado para a geração de seqüências que exercitam as transições de estados (Kansomkeat e Rivepiboon, 2003).

A abordagem de Fröhlich e Link prevê a realização de um mapeamento dos elementos presentes em documentos de casos de uso para uma máquina de estados UML (Fröhlich e Link, 2000). Baseado na máquina de estados, métodos de planejamento (*AI Planning*) são aplicados para derivar conjuntos de teste que proporcionem um nível de cobertura. As respostas esperadas do sistema, assim como os dados de teste concretos devem ser definidos manualmente.

Bertolino e Marchetti desenvolvem uma abordagem voltada para contextos industriais que requerem resultados eficazes rapidamente, mesmo quando o software encontra-se modelado de forma parcial ou incompleta (Bertolino e Marchetti, 2005). Por meio de uma série de passos, diagramas de estados e diagramas de seqüência são combinados em um modelo com mais informações para o teste, chamado de “diagrama de seqüência razoavelmente completo” (SEQrc). Este modelo é utilizado em uma metodologia incremental para construção sistemática de casos de teste.

Fantinato e Jino propõem a extensão das máquinas de estados finitos para a representação de informações de fluxo de dados (Fantinato e Jino, 2003). O modelo proposto Máquina de Estados Finitos Estendida (MEFE) torna possível que cenários de teste sejam selecionados pela identificação de elementos requeridos semelhantes aos definidos em critérios de teste estrutural baseado em análise de fluxo de dados. O modelo MEFE se diferencia do MEF essencialmente pela definição de aspectos de fluxo de dados, como variáveis, predicados e comandos envolvendo variáveis. Ocorrências de variáveis na MEFE podem ser: uma definição (atribuição de valores); um uso predicativo (uso valores em predicados); ou um uso computacional (uso valores em outros comandos, como em um comando de saída, ou do lado direito em uma atribuição). Critérios de teste são propostos para selecionar cenários de teste que exercitem associações entre definições e usos de variáveis. Um estudo de caso mostra a aplicação da abordagem em uma aplicação da área

de telecomunicações e utilizando a ferramenta apoio ao teste estrutural POKE-TOOL (Chaim, 1991) para seleção dos elementos requeridos a partir da MEFE.

Hemmati et al. destacam o problema prático de que no nível de teste de sistema o TBM pode gerar conjuntos de teste muito grandes, tornando a abordagem inviável em certos casos, mesmo usando critérios de teste menos exigentes (ex; cobertura de transições) (Hemmati et al., 2010). Para reduzir o tamanho dos conjuntos de teste, é proposta uma técnica de seleção de casos de teste baseada em similaridade. Aplicada em conjuntos de teste gerados automaticamente de máquinas de estado UML, a técnica proposta busca minimizar a similaridade entre os casos de teste selecionados. Esta similaridade é avaliada por meio de “guardas” associadas às transições de estado e é utilizada por um Algoritmo Genético (Goldberg, 1989) que seleciona amostras de casos de teste com mínima similaridade entre pares. A premissa da proposta é que reduzir a similaridade entre os casos de teste leva a um maior número de defeitos revelados.

Outros trabalhos voltados ao contexto industrial incluem os seguintes. O projeto AGEDIS para a criação de metodologia e ferramentas para a automação da geração e execução de testes dirigida por modelos para sistemas distribuídos (Hartman e Nagin, 2004). Sarma et al. descrevem um estudo de caso sobre a seleção e utilização de duas ferramentas comerciais de teste baseado em modelos, uma baseada em *statecharts* UML e outra em um modelo de comportamento gerado a partir de componentes específicos de modelagem no ambiente .NET (Sarma et al., 2010). Abordagens de teste usando *stacharts* são apresentadas por Kansomkeat e Rivepiboon (Kansomkeat e Rivepiboon, 2003). Santiago Junior et al. apresentam uma abordagem para geração de dados de teste utilizando *statecharts* e Z para o teste de software embarcado em satélites (Santiago Junior et al., 2010). Bringmann e Krämer abordam o teste baseado em modelos de sistemas automotivos usando uma abordagem chamada de Teste de Partição de Tempo (*Time Partition Testing – TPT*) para a seleção de casos de teste (Bringmann e Krämer, 2006).

Dois modelos se destacam no apoio ao projeto de casos de teste. Diagramas de Casos de Uso são extensivamente utilizados para representar requisitos de sistemas, o que os tornam candidatos para o projeto do teste de sistemas. Modelos baseados em estados são

bem embasados teoricamente e adequados à automação de tarefas no projeto do teste. São aplicáveis a sistemas caracterizados por estados que mudam conforme estímulos fornecidos. A seguir é feita uma breve descrição sobre a aplicação desses modelos no projeto de casos de teste.

Teste Baseado em Casos de Uso

O Teste Baseado em Casos de Uso utiliza os modelos de Casos de Uso – UML, criados na fase de análise de requisitos e de projeto do software, para a seleção de casos de teste (Jacobson, Booch e Rumbaugh, 1999). Casos de Uso são muito usados para especificar o comportamento esperado do sistema do ponto de vista dos atores que interagem com o sistema. Como são elaborados normalmente em etapas iniciais do processo de desenvolvimento, esses modelos são úteis para o projeto antecipado de casos de teste, principalmente os que serão aplicados no teste de sistema e no teste de aceitação do software.

A aplicação do critério de teste é feita nos seguintes passos:

- a) Descrever os fluxos de eventos para o caso de uso. Devem ser descritos tanto o fluxo de eventos básico como os fluxos alternativos e de exceção.
- b) Definir o conjunto de cenários que serão usados nos testes. Cada cenário pode ser formado a partir dos fluxos de eventos identificados bem como das combinações desses fluxos.
- c) Definir casos de teste para cada um dos cenários, ou seja, para cada cenário definir valores de entrada, ações no software que provoquem a execução do cenário e as saídas esperadas.
- d) Executar o software com os dados de teste, avaliar se os resultados produzidos correspondem aos esperados, registrar os resultados e os incidentes observados.

Notar que os passos *a*, *b* e *c* podem ser realizados logo que a especificação dos casos de uso esteja pronta e revisada; não é necessário esperar a conclusão da implementação dos

casos de uso para realizá-los. O passo *d* pode ser realizado em momentos distintos, dependendo do nível do teste realizado: quando os módulos que implementam as funções especificadas nos casos de uso estiverem disponíveis (no caso do teste de unidades e do teste de integração); quando o sistema como um todo estiver integrado e disponível (no caso do teste de sistema); ou ainda, quando o sistema for utilizado no processo de aceitação pelo cliente (no caso do teste de aceitação).

Binder destaca que representações de caso de uso não contemplam informações básicas necessárias ao teste (Binder 2000). É proposta a extensão desses modelos pela determinação dos seguintes aspectos:

- O domínio de cada variável envolvida no caso de uso;
- Os relacionamentos entrada/saída requeridos para essas variáveis;
- A frequência relativa de cada caso de uso; e
- Dependências de sequência entre os casos de uso.

Uma representação estendida dos casos de uso é criada para o teste, incluindo informações sobre as variáveis envolvidas nos casos de uso – chamadas de Variáveis Operacionais; sobre a frequência de execução dos casos de uso e sobre dependências entre casos de uso. A análise necessária à criação desta representação estendida permite, por si só, a identificação de inconsistências e omissões nos casos de uso.

Variáveis Operacionais são entradas, saídas, e condições do ambiente que: 1) resultam em um comportamento significativamente diferente para um ator; 2) abstraem o estado do sistema em teste; ou 3) resultam em uma resposta do sistema significativamente diferente. Estas variáveis tipicamente são visíveis nos limites do sistema, como em interfaces gráficas.

Relações lógicas entre as variáveis operacionais, chamadas de Relações Operacionais, são representadas por meio de tabelas de decisão. Essas tabelas representam em colunas condições e ações, usando as variáveis operacionais. Quando todas as condições em uma coluna são verdadeiras, as respectivas ações são produzidas. Por meio dessas tabelas é

possível selecionar sistematicamente casos de teste. Critérios como Particionamento em Classes de Equivalência e Análise de Valores Limites podem ser empregados para a seleção de valores para exercitar os casos de uso.

Briand e Labiche apresentam uma metodologia para apoiar a derivação de requisitos de teste de sistema a partir de artefatos de análise UML (Briand e Labiche, 2001). Os principais artefatos UML considerados são: diagramas de casos de uso e descrições dos casos de uso; diagramas de seqüência e de colaboração correspondentes a cada caso de uso; e um dicionário de dados que descreve cada classe, método e seus atributos.

A metodologia define passos que consideram os vários diagramas para elaborar os casos de teste a serem combinados em um único conjunto de teste. Os diagramas são transformados em grafos que, por sua vez, são utilizados para geração de expressões regulares representando os elementos requeridos para o teste. Um ponto importante é a definição de um tratamento sistemático para testar seqüências de casos de uso. Os principais passos são descritos a seguir, além desses, a metodologia contempla também a elaboração de oráculos e do ambiente de teste:

- 1) Checar a completude, a correção e consistência dos modelos. Os modelos UML são avaliados em relação à testabilidade, isto é, a facilidade com que eles podem ser utilizados para apoiar as atividades de teste, levando em conta a metodologia proposta;

- 2) Derivar seqüências de dependências entre casos de uso. Os casos de uso representam funcionalidades providas pelo sistema e possuem dependências em relação à seqüência de execução relativas ao domínio de aplicação, além das dependências expressas por relacionamentos do tipo extensão e inclusão. Dependências de seqüência são representadas por meio de diagramas de atividades para cada ator do sistema. Neste diagrama vértices são casos de uso e arcos determinam dependências de seqüência entre eles, eventualmente associadas a condições de execução (condições expressas usando *Object Constraint Language – OCL*). Estes diagramas são representados como grafos, que são analisados para criar expressões regulares relacionadas aos possíveis caminhos no grafo.

3) Derivar requisitos de teste a partir de diagramas de sequência. Diagramas de sequência e diagramas de colaboração descrevem como os casos de uso são realizados pela interação entre objetos. Esses diagramas também modelam alternativas de seqüências de operações, cada uma associada a um cenário de uso no domínio de aplicação. Tem-se deste modo um detalhamento de situações de teste em relação aos casos de uso e seqüências de casos de uso identificadas no passo anterior. Esses diagramas também são representados na forma de expressões regulares formadas por referências a métodos e condições de execução associadas a eles (condições expressas usando OCL).

4) Os requisitos de teste derivados nos passos anteriores e representados como expressões regulares são consolidados. Cada requisito (expressão regular) representa um caminho em um diagrama contendo condições a serem satisfeitas para que o caminho seja executado. Tabelas de decisão são utilizadas para formalizar os requisitos de teste. Cada coluna das tabelas representa uma variação que precisa ser exercitada por algum caso de teste.

O teste estrutural de casos de usos é proposto por Carnielo, Chaim e Jino com dois objetivos: gerar dados de teste a partir da estrutura interna de diagramas de casos de uso; e testar a própria especificação modelada como casos de uso (Carnielo, Chaim e Jino, 2004). Critérios de teste propostos consideram os seguintes relacionamentos: associação entre um ator e um caso de uso; inclusão de um caso de uso por outro caso de uso; extensão de um caso de uso por outro caso de uso; e generalização de um ator para outro ator.

Os critérios estruturais requerem que os relacionamentos definidos acima sejam exercitados por pelo menos um caso de teste, sendo que o significado de “exercitar” depende do tipo de relacionamento. O critério *Todas-Associações-Inclusões-Extensões* requer para um conjunto de teste T e um diagrama de casos de uso D , T deve fazer com que cada relacionamento de: associação, inclusão e extensão em D seja exercitado pelo menos uma vez. O critério *Todas-Combinações-Extensões* requer para um conjunto de teste T e um diagrama de casos de uso D , T deve fazer com que todas as combinações de exercitar e não exercitar os relacionamentos de extensão sejam exercitados pelo menos uma vez. Por exemplo, se em D há um caso de uso A com extensões para dois outros casos de uso (B e C)

será requerido o teste das quatro possíveis combinações exercitar-não-exercitar (E,NE) para as extensões de A para B e de A para C, o que resulta nas seguintes combinações: {(E,E), (E,NE), (NE,E), (NE,NE)}.

Uma ferramenta de cobertura denominada *UCT (Use Case Tester)* apóia a aplicação dos critérios por meio das funcionalidades: determinação dos requisitos de teste; simulação da execução dos casos de teste em uma especificação de casos de uso; e análise de cobertura os casos de teste. Assim como no trabalho anterior, os diagramas de casos de uso são representados internamente como grafos, utilizados nas simulações de comportamento dos casos de uso.

A ferramenta e os critérios foram aplicados em um estudo de caso, o teste de um sistema de cobranças em telefonia móvel. Um conjunto inicial de teste foi aplicado e a cobertura avaliada. Os elementos não exercitados por este conjunto inicial para o critério Todas-Combinações-Extensões levaram à definição de casos de teste adicionais e também à identificação de uma inconsistência semântica na especificação.

Teste Baseado em Máquinas de Estados Finitos

Certos tipos de software podem ser descritos como um conjunto de estados que podem ocorrer durante a execução e por estímulos que acarretam as mudanças de estados. Nesses casos modelos de estados são adequados para apoiar o testador na construção de seqüências a serem usadas como entrada para o software. O software ou sistema exhibe este comportamento “baseado em estado” quando entradas idênticas não são sempre aceitas e, quando são aceitas, podem produzir diferentes resultados.

Máquinas de estados finitos (MEF) são modelos bem embasados teoricamente e que abstraem sistemas identificando eventos (entradas), estados, e ações (saídas). Aplicados inicialmente no projeto e teste de hardware, tais modelos, tem um espaço importante no projeto e no teste de software. Uma MEF pode ser vista como um sistema no qual a saída é determinada pela entrada atual e também pelas entradas anteriores. O efeito dessas entradas anteriores é representado por um estado do sistema. Um exemplo simples é um cadeado

com segredo numérico (sem chaves) em que a abertura do cadeado depende do fornecimento seqüencial nos números que formam o segredo.

Uma MEF pode ser representada como um diagrama de transição de estados na qual os estados são representados por círculos e as transições são representadas por arcos direcionados entre os estados. Cada arco é rotulado com a entrada que gera a transição e a saída que é produzida. Uma MEF pode também ser representada como uma tabela de transição, com estados representados por linhas e as entradas por colunas, para uma descrição completa veja (Simão, 2007).

Formalmente, uma MEF é definida pela tupla $M = (X, Z, S, S_0, f_z, f_s)$, sendo que: X é um conjunto finito, não vazio, de símbolos de entrada; Z é um conjunto finito de símbolos de saída; S é um conjunto finito, não vazio, de estados; S_0 é o estado inicial; $f_z: (S \times X) \rightarrow Z$ é a função de saída; $f_s: (S \times X) \rightarrow S$ é a função de próximo estado (Simão, 2007).

O estado inicial é um estado no qual o primeiro evento é recebido. Uma transição provoca uma mudança de estado no sistema e é caracterizada por um par de estados, o estado de recepção e o estado resultante (que pode ser o mesmo). A máquina pode estar em apenas um estado em um dado momento, o estado ativo ou corrente. Uma transição é realizada a partir de um estado corrente para um estado resultante. O estado final é o estado no qual a máquina para de receber eventos.

Algumas estratégias de teste aplicadas em FSM descritas em (Binder, 2000):

Todas as transições: requer que toda transição seja exercitada pelo menos uma vez. Necessariamente exercita todos os estados, todos os eventos e todas as ações. Seqüências particulares não são requeridas. Esta estratégia é também chamada de TT (Simão, 2007).

Todas as seqüências n-transições: requer que todas as seqüências de transições de n eventos sejam exercitadas pelo menos uma vez. Por exemplo, todas as seqüências 3-transições, requer que todas as seqüências de três estados sejam exercitadas.

Todos os caminhos de ida e volta (all round-trip paths): requer que toda sequencia de transições iniciando e terminando no mesmo estado seja exercitada pelo menos uma vez.

Outras estratégias para exercitar elementos na MEF podem ser encontradas na literatura de linguagens formais e teoria de autômatos, por exemplo:

No Método-w (Chow, 1978) são gerados dois conjuntos de sequências de entrada: P, o conjunto de sequências que cobre cada transição pelo menos uma vez, e T, o conjunto de sequências para identificar qual é o estado da máquina. As sequências a serem utilizadas no teste são geradas pela concatenação de P com T.

O método UIO descrito em (Simão, 2007) considera sequências específicas UIO. Uma sequência UIO (*unique input/output sequence*) de um estado s_j é uma sequência de eventos entrada/saída única a partir desse estado, que permite distinguir o estado s_j de qualquer outro estado de S. O método utiliza essas sequências para gerar um conjunto de dados de teste (conjuntos de sequências de entrada). Neste conjunto, para cada estado s_i , uma sequência de entradas que leva a MEF a partir do estado inicial para s_i e concatenada com a sequência UIO para s_i . Estas duas sequências concatenadas formam um dado de teste.

Definição de Critérios da Técnica de Teste Baseada em Estrutura e descrição da Ferramenta Poke-Tool

Esta Seção retoma os conceitos descritos informalmente nas seções anteriores e define conceitos essenciais para a compreensão e aplicação de critérios de teste estruturais. Para uma descrição mais detalhada e completa, favor consultar (Maldonado, 1991), (Myers, 1979), (Rapps e Weyuker, 1985), (Zhu e Hall, 1993).

Conforme definido informalmente em seção anterior, um *bloco*¹⁵ é uma sequência máxima de comandos tal que, se o primeiro comando for executado, então, necessariamente, todos os demais subseqüentes no bloco, também serão executados. Em um grafo G , representando um software S , um *nó*, é a representação gráfica de um *bloco* e

¹⁵ O primeiro comando de um bloco é o único comando que pode ser diretamente executado após a execução de um comando dentro de outro bloco.

um *Ramo*, ou *arco*, é a representação gráfica de uma possível transferência de controle entre os blocos.

O nó cujo primeiro comando é também o primeiro comando do programa, é conhecido como *nó de entrada*. Um nó cujo último comando é o comando final do programa, é conhecido como *nó de saída*. Um *caminho* é uma seqüência finita de nós (n_1, n_2, \dots, n_k) , $k \geq 2$, tal que exista um *arco* de n_i para n_{i+1} , para $i = 1, 2, 3, \dots, k-1$.

Um *caminho simples* é tal que todos os nós, exceto possivelmente o primeiro e o último, são distintos. Um *caminho é livre de laço* se todos os nós pertencentes a ele são distintos. Um *caminho completo* é um caminho cujo nó início é o nó de entrada e o nó final é o nó de saída.

A análise de fluxo de dados focaliza as ocorrências das variáveis nos programas, podendo ser da forma de *definição* ou *uso* de variável.

Uma *definição* de variável ocorre quando um valor é armazenado em uma posição de memória. Em geral, uma ocorrência de variável é uma definição se ela está: i) no lado esquerdo de um comando de atribuição; ii) em um comando de entrada; ou iii) em chamadas de procedimentos como parâmetro de saída.

A ocorrência de uma variável como *uso* se dá quando a referência a esta variável não a estiver definindo, ou seja, há uma *recuperação* de um valor em uma posição de memória associada a esta variável. Um *uso* de variável pode afetar diretamente uma computação que está sendo realizada, ou permitir que o resultado de uma variável definida anteriormente seja observado – nestes casos, o uso está associado a um nó do grafo do software e é chamando de *Uso Computacional (c-uso)*, ou ainda afetar diretamente o fluxo de controle do programa - este uso está associado a um arco do grafo e é chamando de *Uso Predicativo (p-uso)*.

Um caminho (i, n_1, \dots, n_m, j) , $m \geq 0$ que não contenha nenhuma definição de uma dada variável x nos nós n_1, \dots, n_m é chamado de *caminho livre de definição* com respeito a (c.r.a) x do nó i ao nó j e do nó i ao arco (n_m, j) .

Um nó i possui uma *definição global* de uma variável x se ocorrer uma definição de x no nó i e se existir um caminho livre de definição c.r.a x de i para algum nó ou para algum arco que contenha um uso da variável x . Uma definição de x no nó i é uma *definição local* se existirem usos – associados ao nó – da variável x somente neste nó i e que sucedam esta definição.

Exemplos de definições e usos de variáveis:

- O comando “ $y \leftarrow f(x_1, \dots, x_n)$ ”: contém usos (c-usos) de x_1, \dots, x_n seguido de uma definição de y ;
- O comando “*read* x_1, \dots, x_n ”: contém definições de x_1, \dots, x_n ;
- O comando “*print* x_1, \dots, x_n ”: contém usos (c-usos) de x_1, \dots, x_n ;
- O comando “*if* $p(x_1, \dots, x_n)$ *then* S ” contém usos (p-usos) de x_1, \dots, x_n .

Uma *associação definição-uso (def-uso)*, dada pela tripla (i, j, x) , representa o conjunto de todos os caminhos livres de definição c.r.a x , tendo seu início no nó i , onde a variável foi definida e seu fim no nó j , onde a variável foi utilizada. Uma associação def-uso será coberta – ou exercitada – quando ao menos um desses caminhos for executado.

Um caminho $(n_1, n_2, \dots, n_j, n_k)$ é um *du-caminho c.r.a* variável x se 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 c.r.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 c.r.a x e n_1, n_2, \dots, n_j é um caminho livre de laço.

Um caminho completo é executável se existir alguma associação de valores às variáveis de entrada que cause a execução do caminho. Um caminho (e também um du-caminho) é executável se ele é um sub-caminho de algum caminho completo executável. Uma associação é executável se existir ao menos um caminho executável que a cubra. Um arco ou nó é executável se residir em algum caminho executável.

O impacto da existência de elementos requeridos não executáveis no teste estrutural é que, ao se aplicar o critério para o teste de um software S , alguns dos elementos requeridos pelo critério podem ser não executáveis. O problema é que não existe algoritmo de

propósito geral que decida se um dado caminho é executável ou não. Deste modo, após a execução de dados de teste, e ao se identificar elementos requeridos não exercitados, uma parte desses requisitos pode ser não executável. Neste caso, cabe ao testador a tarefa de inspecionar os requisitos não executados para identificar se eles são ou não executáveis. Discussões sobre a questão da não executabilidade no teste estrutural e suas implicações podem ser encontradas em (Vergilio, 1992).

Considerando-se um software S e seu respectivo grafo G , e considerando-se um conjunto de caminhos P nesse grafo.

Podem ser definidos os critérios baseados em análise de fluxo de controle:

- **Critério Todos os Nós** requer que a P passe, ao menos uma vez, em cada nó de G ; ou seja, que cada comando de S seja executado pelo menos uma vez;
- **Critério Todos os Ramos:** requer que a P passe, ao menos uma vez, em cada arco de G ; ou seja, que cada desvio de fluxo de controle de S seja executado pelo menos uma vez;
- **Critério Todos os Caminhos:** requer a execução de todos os caminhos de G .

Podem ser definidos os critérios baseados em análise de fluxo de dados:

- **Critério Todas as Definições.** P satisfaz o critério se para todo nó i do grafo G e toda variável x definida neste nó, P incluir um caminho livre de definição c.r.a x , executável, para algum c -uso ou p -uso (não importando qual) de x a partir do nó i . Portanto, por este critério, exige-se um uso qualquer, de um tipo qualquer, para todas as definições existentes no programa, ou seja, todas as definições de variáveis serão utilizadas pelo menos uma vez.
- **Critério Todos os Usos.** P satisfaz o critério se para todo nó i do grafo G e toda variável x definida em i , P inclui um caminho livre de definição c.r.a x , executável de i para todos os c -usos de x a partir de i , e para todos os p -usos de x , a partir de i . Tal critério exige portanto que para cada definição de variável, todos os usos de valores atribuídos nesta definição sejam testados

(independentemente do tipo de uso). Note-se que para exercitar cada associação, basta executar um único caminho que a cubra.

- **Critério Todos os du-Caminhos.** P satisfaz o critério se para todo nó i , e para cada variável x definida no nó i , P inclui todos du-caminhos executáveis com respeito a x existentes entre o nó i e todos os *usos* de x (sejam *p-usos* ou *c-usos*), ou seja, não se exige “apenas” que todas as associações def-uso sejam cobertas. Exige também que cada associação seja coberta por todos os du-caminhos possíveis.

Exemplos de aplicação desses critérios podem ser encontrados em (Maldonado, 1991), Capítulo 4. Uma descrição completa de critérios baseados em análise de fluxo de dados encontra-se em (Barbosa et al., 2007).

A aplicação de critérios estruturais em ambientes profissionais requer a utilização de ferramentas de apoio. Estas ferramentas são tipicamente voltadas a uma linguagem de programação específica e apóiam o testador em atividades típicas do teste estrutural, tais como: análise estática do código fonte; geração de grafos de fluxo de controle e grafos de chamadas de módulos; definição de elementos requeridos para o teste; e análise de cobertura atingida no teste.

Lutz descreve genericamente a utilização de ferramentas no teste de software (Lutz, 1990). Yang et. al. apresentam uma revisão sobre ferramentas de teste de apoio ao teste estrutural (Yang et. al. 2006); Poston e Sexton apresentam diretrizes para avaliação e seleção de ferramentas de teste (Poston e Sexton, 1992). Ferramentas de teste de código aberto e discussões associadas são disponíveis em (OSSTT¹⁶, 2011)

A ferramenta de teste POKE-TOOL, desenvolvida na FEEC/Unicamp em colaboração com o ICMSC/USP apóia a aplicação do teste estrutural (Chaim, 1991), (Chaim et al., 1995). A ferramenta apoia o uso de critérios de teste Potenciais Usos, descritos na Seção anterior, e os critérios baseados em análise de fluxo de controle e os

¹⁶ <http://www.opensourcetesting.org/>

baseados em análise de fluxo de dados descritos acima. Algumas características da POKE-TOOL:

- Está disponível em ambiente UNIX e permite utilização por interface gráfica ou scripts Shell;
- Incorpora o conceito de seção de testes, que envolve as atividades: análise estática; preparação para o teste; submissão de casos de teste; avaliação de casos de teste e gerenciamento dos resultados de teste.
- Em uma fase estática a ferramenta realiza a análise do código fonte; a instrumentação do código com instruções que produzem um rastreamento do caminho executado, gerando um programa instrumentado; e a determinação de elementos requeridos segundo o critério de teste selecionado;
- A fase dinâmica consiste em gerar o executável do programa instrumentado, executar os casos de teste, e avaliar os resultados do teste de acordo com o critério de teste selecionado. Esta avaliação gera como informação: o conjunto de elementos exercitados; o conjunto de elementos não exercitados; a cobertura atingida; entradas e saídas do programa e caminhos executados pelos dados de teste.

Procedimento de Teste na Análise de Mutantes

O processo de teste de um programa p usando a Análise de Mutantes é realizado do seguinte modo.

Um conjunto de *programas mutantes* p' é gerado por meio de pequenas mudanças sintáticas no programa original p . Cada mutante p' difere de p apenas por uma única mudança sintática realizada. Por exemplo: um operador relacional pode ser substituído por outro operador relacional (ex: “<” substituído por “>”); uma variável pode ser substituída por outra variável (ex: variável real “x” substituída por variável real “y”); um operador aritmético pode ser substituído por outro operador aritmético (ex: “+” substituído por “-”), etc. Regras de transformação para gerar mutantes a partir do programa original são

chamadas de *operadores de mutação*. Tipicamente tais operadores modificam variáveis e expressões por meio da substituição, inserção, ou deleção de elementos.

A aplicação de um operador de mutação em um programa normalmente resulta na geração de vários mutantes, pois o programa pode conter várias entidades que estão no domínio do operador. Por exemplo, no caso de substituição de operadores relacionais, cada ocorrência de operador relacional em p é tratada distintamente para geração de mutantes. Ainda no exemplo, cada ocorrência de operador (ex: “>”) pode gerar mais de um mutante, cada um com uma substituição diferente (“=”, “≥”, “<”, etc.).

Após a geração dos programas mutantes, um conjunto de dados de teste T é executado no programa p e as saídas produzidas são checadas em relação ao esperado. Em caso de falhas, p deve ser corrigido. Cada mutante p' é então executado com cada dado de teste t de T . Se o resultado da execução de p' é diferente do resultado da execução de p para algum dado de teste t , então o mutante p' é considerado *morto*, caso contrário, p' é considerado *vivo*. Ao longo do teste, a execução dos dados de teste em T é realizada apenas nos mutantes que ainda estão vivos.

Após a execução dos dados de teste podem existir mutantes vivos. O conjunto T pode então ser aprimorado com dados de teste adicionais para matar esses mutantes sobreviventes.

Existem mutantes que não podem ser mortos porque eles produzem sempre a mesma saída do programa original, são chamados de *mutantes equivalentes*. Apesar de serem sintaticamente diferentes, eles são funcionalmente equivalentes ao programa original. Considere, por exemplo, um comando de repetição `for(int i = 0; i < 0; i++) { ... }` alterado para `for(int i = 0; i != 0; i++) { ... }` em um mutante. Neste caso, se o valor de i não é alterado por algum comando dentro do laço *for*, então o número de iterações do laço será idêntico para o programa e para o mutante, assim como a saída final produzida por ambos. A detecção automática de mutantes equivalentes é em geral impossível visto que detectar a equivalência de dois programas é um problema indecidível.

O *escore de mutação* é uma medida de cobertura do critério análise de mutantes e denota o nível de adequação de um conjunto de dados de teste sob a perspectiva da Análise de Mutantes. Dado um programa P e um conjunto de dados de teste T , o escore de mutação $MS(P, T)$ é dado por:

$$MS(P, T) = \frac{DM(P, T)}{M(P) - EM(P)}$$

Sendo que $DM(P, T)$ é o conjunto de mutantes mortos pelo conjunto de dados de teste T ; $M(P)$ é o número total de mutantes gerados a partir do programa P ; e $EM(P)$ é o número de mutantes gerados que são equivalentes a P .

Considerações Sobre a Aplicação da Análise de Mutantes

Um obstáculo à adoção ampla desta técnica é o alto custo computacional para executar um número muito grande de mutantes com os dados de teste. Outro custo importante é o esforço humano para a avaliação de mutantes equivalentes e para a avaliação das saídas produzidas pelo programa em teste – este último comum a qualquer técnica de teste.

Redução de Custo de Aplicação

Abordagens para a redução de custo da Análise de Mutantes têm sido propostas, seguindo duas linhas: redução do número de mutantes gerados e redução do custo de execução (Jia e Harman, 2010).

Abordagens de redução do número de mutantes buscam reduzir o número de mutantes gerados sem que haja uma perda significativa na eficácia do teste. Este problema pode ser definido como segue: considerando um conjunto de mutantes M e um conjunto de dados de teste T , a redução busca encontrar um subconjunto de mutantes M' , definido como M' , tal que o escore de mutação do conjunto T aplicado aos mutantes em M e o escore de mutação do conjunto T aplicado aos mutantes em M' tenham valores próximos.

Abordagens de redução do número de mutantes incluem a Amostragem de Mutantes em que um pequeno subconjunto de mutantes é selecionado aleatoriamente do conjunto total de mutantes gerados (Acree, 1980). A Mutação Seletiva realiza a redução do número de mutantes pela redução do número de operadores de mutação, omitir operadores que geram a maioria dos mutantes foi uma proposta. Variações desta abordagem consideram a omissão de dois operadores, de quatro operadores, ou seis operadores (Mathur, 1991), (Offutt et al., 1993). Um procedimento para a seleção sistemática de operadores de mutação é proposta em (Barbosa et al, 2001).

Abordagens para a redução do custo de execução de mutantes seguem as seguintes linhas: o uso de Mutação Fraca e a otimização do tempo de execução.

Conforme já descrito, a abordagem tradicional da análise de mutantes, referida como *Mutação Forte* requer, para que um mutante p' seja considerado morto, que o programa p e o mutante p' , produzam resultados finais distintos. A *Mutação Fraca* é uma abordagem alternativa à tradicional (Howden, 1982). Supondo que um programa p é formado por um conjunto de componentes $\{c_1, \dots, c_n\}$ e que um mutante p' é criado pela mudança de um componente específico c_m , o mutante p' é considerado morto se alguma execução do componente c_m em p é diferente do mutante p' . Deste modo, ao invés de checar os mutantes após a execução completa do programa, os mutantes precisam ser executados e avaliados apenas até imediatamente após o ponto em que está o componente que sofreu mutação.

Níveis de Teste e Objetos de Aplicação

A aplicação da análise de mutantes, na maioria dos casos, é feita no código fonte do programa, tanto no nível de teste de unidades quanto no nível do teste de integração. No nível do teste de unidades, como já descrito, os mutantes representam defeitos inseridos pelos programadores ao implementar as unidades. No nível de integração, também referida como mutação de interfaces (Delamaro, Maldonado e Mathur, 2001), os mutantes são projetados para representar defeitos de integração, relacionados à conexão entre as unidades e às interações que ocorrem nestas conexões. Em geral, as conexões são representadas por

chamadas de procedimentos e de funções e as interações ocorrem por meio de parâmetros trocados e de variáveis globais.

Abordagens análise de mutantes que consideram a especificação e projeto do software são referidas como Mutação de Especificação. Os defeitos nestes casos são inseridos em modelos que representam o software, por exemplo, máquinas de estados ou expressões lógicas são alteradas, gerando “mutantes de especificação”. O teste de mutação em especificações na forma de máquinas de estados finitos (MEF) é abordado por Fabri et al. (Fabri et al., 1999 a). Os operadores de mutação representam defeitos relacionados a estados, eventos e saídas produzidas pela MEF. O teste de mutação utilizando modelos *Statecharts* e Redes de Petri também foi abordado (Fabri et al., 1999 b). Nestes casos o mutante é considerado morto quando o resultado da execução do modelo mutante é diferente do resultado do modelo original.

Ferramentas de Apoio

A aplicação do teste de mutantes requer o uso de ferramentas de apoio. Jia e Harman apresentam informações sobre 36 ferramentas, incluindo ferramentas industriais, acadêmicas, e de código aberto (Jia e Harman, 2010). DeMillo e Offutt abordam a automação da geração de dados propondo técnicas para representar algebricamente as restrições a serem satisfeitas para exercitar mutantes (DeMillo e Offutt, 1991). A ferramenta Godzilla que implementa as técnicas foi utilizada em experimentos de avaliação (DeMillo e Offutt, 1993).

Delamaro e Maldonado descrevem a ferramenta Proteum que apóia o teste de mutação em programas escritos em C (Delamaro e Maldonado, 1996). Maldonado et al. descrevem uma família de ferramentas para apoiar o teste de mutantes (Maldonado et al., 2000). A família, também referida como Proteum, é composta por ferramentas para diferentes níveis e abordagens de teste, incluindo: o teste de unidades de programas C; o teste de integração de programas C usando a mutação de interface; o teste de mutação para Máquinas de Estados Finitos, *Statecharts* e Redes de Petri. Essencialmente estas ferramentas apóiam a realização de atividades para:

- Tratar casos de teste, o que envolve executar, incluir, excluir, habilitar e desabilitar casos de teste;
- Tratar mutantes, o que envolve criar, selecionar, executar e analisar mutantes;
- Avaliar a adequação, o que envolve determinar o escore de mutação e gerar relatórios.

Apêndice B – Geração de Dados de Teste: detalhamento das abordagens

Este apêndice complementa o conteúdo do Capítulo 3 – Geração de Dados de Teste. São descritos outros trabalhos não abordados no capítulo e são fornecidos detalhes das técnicas para geração de dados de teste. Buscou-se não incluir neste apêndice conteúdo já apresentado no Capítulo 3, alguns trechos são replicados apenas por questão de clareza e definição de contexto.

Geração de dados de teste com uso de execução simbólica: outros trabalhos

A seguir são descritos brevemente trabalhos que utilizam execução simbólica.

Clarke propõe uma ferramenta com as seguintes funcionalidades principais: Gerar dados de teste para provocar a execução de um caminho no programa; detectar alguns caminhos não executáveis e criar uma representação simbólica das variáveis de saída do programa como função das variáveis de entrada. São gerados dados de teste apenas quando as restrições associadas ao caminho são lineares. A geração de dados não é possível também na presença de estruturas do tipo vetor (Clarke, 1976).

Ramamoorthy et al. propõem uma abordagem bastante semelhante à anterior, implementada no protótipo CASEGEN (Ramamoorthy et al., 1976). É feito o processamento simbólico do programa, resultando em um conjunto de equações e inequações em função das variáveis de entrada. Para a geração de dados de entrada é utilizada a “tentativa e erro sistemática”. A abordagem segue uma série de passos: Inicialmente as restrições estruturais são descritas em forma conjuntiva. As variáveis de entrada são organizadas e associadas às restrições que as incluem. São atribuídos valores aleatórios de entrada, considerando uma variável por vez. Sempre que não é satisfeita

alguma cláusula é feito um *backtracking* e são atribuídos novos valores. Tem-se um processo iterativo que termina quando todas as variáveis forem definidas para satisfazer as cláusulas ou, caso estes valores não sejam encontrados, tem-se a falha na geração de dados.

Howden apresenta o Sistema DISSECT que apoia a avaliação simbólica de programas em Fortran (Howden, 1975), (Howden, 1977). Dois arquivos são entradas do sistema: um contendo o programa a ser testado e outro uma lista de comandos que controlam a avaliação simbólica, permitindo a realização do teste. Os comandos de entrada permitem que valores reais ou simbólicos sejam atribuídos às variáveis durante a execução do programa. Comandos de saída permitem que os valores das variáveis ou dos predicados associados aos caminhos do programa sejam fornecidos ao testador. É introduzida uma abordagem para o tratamento de vetores baseada em lista de valores simbólicos atribuídos e tratamento de referências ambíguas. É feita também uma análise da adequação do teste simbólico às classes de defeitos. Não é abordada a geração automática de dados que executem os caminhos processados simbolicamente.

Boyer et al. apresentam um sistema experimental – o SELECT, compatível com um subconjunto da linguagem LISP, que faz o processamento dos caminhos do programa com os seguintes objetivos: gerar dados de teste; prover valores simbólicos para as variáveis do programa como resultado da execução dos caminhos; e, provar a correção de um caminho com respeito a asserções informadas pelo usuário (Boyer et al., 1975). A geração dos dados de teste é feita por um algoritmo de gradiente conjugado. Este algoritmo busca minimizar uma função potencial construída a partir das inequações simbólicas geradas. A abordagem requer interação com o usuário, além de não terminar garantidamente.

Abordagem dinâmica para a geração de dados de teste: outros trabalhos

Gallagher e Narasimhan apresentam o ADTEST, um sistema para geração automática de dados de teste para programas escritos em Ada (Gallagher e Narasimhan, 1997). A abordagem utilizada baseia-se na execução do programa e na aplicação de otimização numérica.

É definida a função:

$$F(x, w) = \sum_{i=1}^n G(g_i(x), w_i, type_i)$$

Onde w_1, \dots, w_n são pesos positivos e o termo $G(g_i(x), w_i, type_i)$ representa a restrição imposta pelo predicado i do caminho. Valores $g_i(x)$ são baseados nos operadores relacionais $type_i$ envolvidos nos predicados e são determinados dinamicamente pela execução do programa instrumentado. A função G assume valores pequenos se os predicados são satisfeitos e valores maiores, caso contrário. Uma técnica de otimização é empregada para minimizar (x, w) . Durante a busca, à medida que os predicados são satisfeitos, novos predicados do caminho e respectivos valores $G(g_i(x), w_i, type_i)$ são considerados na otimização.

Gupta et al. utilizam uma abordagem baseada em execução denominada Método de Relaxamento Iterativo (Gupta et al., 1998). O objetivo é gerar dados de teste que provoquem a execução de um caminho pré-definido. O método inicia com a aplicação de valores de entrada gerados aleatoriamente; se o caminho desejado não é executado então a entrada é refinada de forma iterativa.

Para aplicação do método são derivadas duas representações para cada predicado do caminho: o *slice* do predicado é o subconjunto de entradas e comandos de atribuição que precisam ser executados para a avaliação do predicado. Esta é uma representação exata da função computada pelo predicado. Usando esta representação é derivada a *representação linear aritmética* do predicado em termos das variáveis de entrada do programa. Esta representação é gerada para permitir a aplicação de técnicas de análise numérica. Se a função computada pelo predicado é linear com respeito às variáveis de entrada, então a representação linear aritmética é exata; senão, esta representação aproxima o valor da função associada ao predicado na sua vizinhança.

Estas duas representações são utilizadas para refinar os valores de entrada iniciais a fim de obter a entrada desejada do seguinte modo: se a execução do *slice* do predicado determinar que ele não foi avaliado como desejado, a avaliação da função computada pelo predicado provê um valor chamado *resíduo do predicado*. Este resíduo é a quantidade que

o valor da função precisa mudar para que o predicado seja avaliado como desejado. Utilizando este valor e a representação linear aritmética do predicado é derivada a *restrição linear nos incrementos* para entrada atual. Esta restrição é derivada para cada função de predicado do caminho. Todas as restrições são então resolvidas simultaneamente, usando um método para solução de sistemas lineares, para computar os incrementos na entrada atual.

Neste método, caso as funções de predicado sejam todas lineares em função das variáveis de entrada, é gerada a entrada desejada em uma iteração ou é garantido que o caminho é não executável. Entretanto, basta que alguma função de predicado seja não linear para que a geração falhe nesta tentativa. Nestes casos podem ser usados diversos refinamentos sucessivos para que a avaliação do predicado seja a desejada. Em cada refinamento tem-se a execução dos slices computados para todos os predicados do caminho e a computação dos valores: representação linear aritmética e resíduo, para os todos os predicados.

Teste Baseado em Buscas com Ênfase na Implementação

Jones et al.

Jones et al. aplicam um Algoritmo Genético com o objetivo de exercitar ramos no software (Jones et al., 1996).

A função de ajuste é baseada nos predicados a serem satisfeitos para o teste de ramos e direciona a busca para encontrar valores de entrada que provoquem a execução de um determinado ramo do programa. O gerador de dados de teste concentra-se em um ramo por vez, e o foco é mudado automaticamente para os próximos ramos até que todos tenham sido tratados.

Os operadores genéticos aplicados são recombinação e mutação simples. A recombinação realiza a troca de bits entre os strings (recombinação uniforme) com probabilidade em torno de 0.5. Na mutação bits são selecionados aleatoriamente e são

alterados de 0 para 1 ou vice-versa. A probabilidade de mutação é inversamente proporcional ao comprimento dos strings.

Resultados experimentais em termos de execuções do programa em teste (ou analogamente avaliações de fitness) para atingir a cobertura completa de ramos foi comparado ao número de execuções necessárias com o teste aleatório. Em geral o número de execuções requeridas pelo algoritmo genético foi expressivamente menor do que o teste aleatório. Por exemplo, para o programa que classifica triângulos o algoritmo genético e o teste aleatório requereram 18000 e 16300 execuções respectivamente.

Michel et al.

Michel et al (2001) definem o problema da geração de dados de teste como “encontrar dados de entrada que satisfaçam um critério de adequação definido”. Uma ferramenta chamada GADGET (Genetic Algorithm Data GEneration Tool) implementa a abordagem para a geração de dados proposta. A geração de dados é baseada em um Algoritmo Genético, que tem como objetivo encontrar conjuntos de dados de teste que satisfaçam o critério Cobertura de Condições e de Decisões.

O problema da geração de dados de teste é reduzido a um problema de minimização de funções: para encontrar o dado de teste desejado, é necessário encontrar um valor de x que minimize $E(x)$. A função $E(x)$ é chamada de função objetivo (ou fitness) e informa ao gerador de dados de teste o quão próximo ele está de alcançar o seu objetivo

A tabela a seguir mostra a função de ajuste utilizada dependendo do tipo de decisão.

Tabela 12. Função objetivo de Michel et al (2001)

<i>decision type</i>	<i>example</i>	<i>fitness function</i>
inequality	<code>if (c >= d) ...</code>	$\mathfrak{F}(x) = \begin{cases} d-c, & \text{if } d \geq c; \\ 0, & \text{otherwise} \end{cases}$
equality	<code>if (c == d) ...</code>	$\mathfrak{F}(x) = d - c $
true/false value	<code>if (c) ...</code>	$\mathfrak{F}(x) = \begin{cases} 1000, & \text{if } c = \text{FALSE}; \\ 0, & \text{otherwise} \end{cases}$

O AG representa as variáveis de entrada como um string de bits e utiliza o operador de recombinação uniforme e a mutação simples. Além deste AG padrão os autores implementaram também um gerador aleatório, um gerador baseado em gradiente descente e um algoritmo genético diferencial, que utiliza um operador de recombinação mais sofisticado.

A aplicação do GADGET para geração de dados de teste para um conjunto de programas simples em um software real de controle de voo com 75 condições e 2046 linhas de código (chamado b737). Dentre os geradores implementados o AG padrão obteve um melhor desempenho. Resultados para o programa b737 mostram que o AG padrão atinge um nível de cobertura de requisitos expressivamente superior ao alcançado usando geração aleatória de dados.

Pargas et al.

Pargas et al. apresentam uma técnica orientada a objetivos para a geração automática de dados de teste. Assim como nos trabalhos anteriormente descritos, um AG é utilizado para a geração de dados (Pargas et al., 1999). O Grafo de Dependências de Controle do programa em teste (GDC) é utilizado para direcionar a busca do AG. Este grafo é definido a partir do Grafo de Fluxo de Controle do programa (GFC) e explicita relações de pós-dominância entre nós existentes neste grafo. Essencialmente, um caminho acíclico no GDC, a partir do nó de início do grafo até um nó N, contém um conjunto de predicados que precisam ser satisfeitos por um dado de teste para que N seja executado.

A Figura 36 mostra um programa exemplo com os nós identificados por números, são mostrados o Grafo de Fluxo de Controle do programa e o Grafo de Dependência de Controle. Observa-se, por exemplo, que os nós 4 e 5 do GFC são dependentes de controle do nó 3, que por sua vez é dependente de controle do nó 2. Os nós 1, 2 e 6 são dependentes de controle apenas do nó de entrada do programa.

```

Program Example
  integer i, j, k
1.  read i, j, k
2.  if (i < j)
3.    if (j < k)
4.      i = k;
    else
5.      k = i;
    endif
  endif
6.  print i, j, k
end Example

```

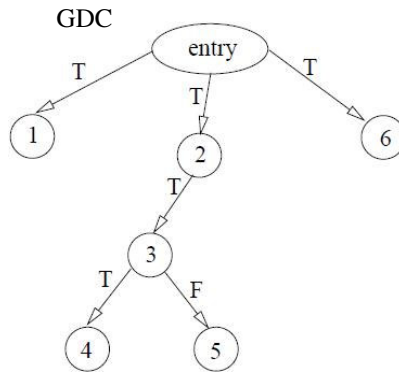
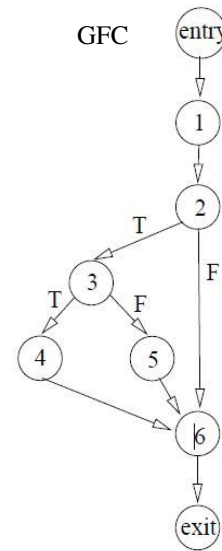


Figura 36. Programa exemplo, GFC e GDC

Uma avaliação empírica desta abordagem foi realizada com a ferramenta aplicada para a geração de dados para exercitar nós. Foram utilizados programas simples que aceitam como entradas variáveis do tipo inteiro. Em comparação com a geração aleatória de dados, a GenerateData teve eficácia equivalente para programas muito simples e mostrou-se superior para, por exemplo o programa tritype, atingindo 100% cobertura com um numero expressivamente menor de execuções do programa em comparação com a geração aleatória.

Wegener et al.

Wegener et al. abordam o uso de meta-heurísticas para o teste de software embarcado (Wegener et al., 2002).

Um ambiente para o teste evolutivo desenvolvido é formado por seis componentes: um *parser* para a análise dos programas em teste; uma interface gráfica para a especificação do domínio de entrada dos programas; um *instrumentador* de programas que permite capturar os elementos exercitados pelos dados de teste; um gerador de *drivers* para executar os programas em teste; um módulo controlador do teste que considera sequencialmente cada objetivo da geração de dados; e um *toolbox* de algoritmos evolutivos para a geração de dados de teste.

Windisch et al.

Windisch et al. utilizam Otimização baseada em enxame de partículas para o teste estrutural (Windisch et al., 2007).

Uma ferramenta de apoio ao teste foi implementada, incluindo módulos para a instrumentação do programa em teste, para a otimização de dados de teste. Além do CL-PSO (baseado em partículas) um Algoritmo Genético também foi implementado, como alternativa para a otimização de dados de teste.

O algoritmo CL-PSO recebe tipos e faixas de valores dos parâmetros de entrada e produz como saída um conjunto de teste com objetivo de atingir alta cobertura estrutural em relação ao critério de teste selecionado (exercitar ramos ou exercitar comandos do programa).

Para cada requisito a ser exercitado (ramo ou comando do programa) uma busca independente é realizada pelo algoritmo CL-PSO, que procura pelo dado de teste que exercite o requisito.

Resultados da aplicação dos algoritmos para programas (alguns artificiais e outros módulos de controle automotivo) que, em geral, a taxa de sucesso para encontrar dados que

exercitem os requisitos foi superior para o algoritmo baseado em partículas, comparado ao AG.

Abreu et al.

Abreu, Martins e Souza abordam a geração de dados de teste para executar caminhos em programas (Abreu, Martins e Souza, 2007).

A função objetivo utilizada NEHD (Normalized Extended Hamming Distance) foi definida por Lin e Yeh e compara dois caminhos de programas retornando um valor de similaridade entre eles (Lin e Yeh, 2001). Deste modo, dados de teste que executam caminhos mais próximos do caminho desejado (segundo a medida NEHD) recebem um valor maior para a função objetivo. Uma breve descrição de como a medida NEHD é calculada é fornecida em (Abreu, Martins e Souza, 2007).

Supondo o caminho desejado representado pelos ramos *abcd* e o caminho executado pelos ramos *be*. O valor objetivo é calculado com o uso da distância hamming aplicada em conjuntos definidos como: *conjunto de interseção (IS)*, que contém ramos que aparecem em apenas um caminho (neste caso ramos a,c,d,e); e *conjunto união (US)*, que contém ramos que aparecem em ambos os caminhos (neste caso a,b,c,d,e). A *similaridade de primeira ordem* entre os caminhos é dada por: $1 - (|IS|/|US|)$, neste caso $1 - (4/5) = 0.2$. A *similaridade de segunda ordem* envolve todos os pares distintos de ramos contíguos nos caminhos, neste caso, os pares de ramos do caminho desejado são {ab, bc, cd} e do caminho executado {be}, |IS| e |US| serão iguais, a similaridade de segunda ordem é zero e o cálculo termina. Caso esta similaridade fosse maior que zero o calculo continuaria em níveis maiores de similaridade. O valor final de similaridade segundo a métrica NEHD faz uma ponderação, associando diferentes pesos aos valores de similaridade das diferentes ordens.

A eficácia do algoritmo OGE foi avaliada no teste de sete programas simples em Java. Como base de comparação foi utilizado um Algoritmo Genético Simples (SGA) e foram analisados o percentual de cobertura de caminhos atingida, além do número de execuções e o tempo gasto durante o processo de geração de dados. Os resultados sugerem

que o desempenho do OGE é comparável ao SGA, superando este algoritmo em programas com maior complexidade.

Tonella

Tonella aborda o teste de programas orientados a objetos (Tonela, 2004). Os casos de teste para métodos de classes incluem a criação de um ou mais objetos, as mudanças de estados internos e a invocação do método em teste. O AG utilizado para a geração de dados é baseado no definido em (Pargas et. al., 1999).

Foi implementada uma ferramenta e foi feita uma avaliação empírica da abordagem proposta para o teste de programas Java. Esta ferramenta inclui módulos para: a instrumentação de programas Java; identificação de assinaturas de métodos; criação de indivíduos; geração de dados de teste; e execução do teste. O objetivo da geração de dados foi o de maximizar a cobertura de ramos em um conjunto de classes da biblioteca padrão Java. Resultados para cada classe são fornecidos em termos de nível de cobertura de ramos atingida, número de casos de teste, e número de iterações da busca. Em geral foram obtidos altos valores de cobertura com um número pequeno de casos de teste; no entanto estes resultados não foram comparados com alguma outra técnica (e.g., a geração aleatória).

Teste Baseado em Buscas com Ênfase em Modelos e/ou na Especificação

Zhao et al.

Zhao et al. apresentam um sistema para a geração automática para caminhos de transições executáveis em EFSM (Zhao et al., 2010). Os autores também realizam uma avaliação empírica da influência de características dos modelos na eficiência da geração de dados baseada em busca. O modelo EFSM é análogo aos definidos anteriormente e é caracterizado por: um conjunto finito de estados; um conjunto de variáveis internas; um conjunto de entradas; um conjunto de saídas transições; e um conjunto de transições. A transição de um estado fonte para outro estado alvo ocorre quando as entradas fornecidas satisfazem uma condição. Esta condição define, na forma de um predicado, um conjunto de

expressões lógicas em termos de variáveis internas e entradas a serem satisfeitas para que a transição ocorra.

Um Algoritmo Genético busca por um conjunto de dados de entrada que provoque a execução de um caminho completo na EFSM. Um caminho p é dado por uma sequência particular de estados s_1, s_2, \dots, s_m . Cada transição de estado está associada a uma condição c_i e ações a_i . Um indivíduo é uma lista de valores de entrada $x = (x_1, x_2, \dots, x_n)$ correspondentes a todos os parâmetros dos eventos (e_1, e_2, \dots, e_m) na ordem em que eles aparecem. Se a sequência de eventos e_i e de parâmetros x_i determina a execução das transições do caminho p e torna válidas as condições c_i de cada transição, então x é a solução para o caminho p . Para que p seja executado é necessário, portanto, que cada restrição imposta pelo caminho seja resolvida e que os predicados anteriores no caminho permaneçam resolvidos durante o processo de busca.

A função objetivo utilizada baseia-se nos componentes, o nível de aproximação (approach level) e distância de ramo normalizada. O nível de aproximação avalia o quão próximo o indivíduo está do caminho alvo, enquanto a distância de ramo mede o quão próximo a primeira condição não satisfeita está de ser verdadeira. Trata-se de uma função objetivo semelhante a utilizada no teste estrutural para exercitar objetivos (Wegener et al., 2002). ou caminhos (Bueno e Jino, 1999). O procedimento de seleção ocorre de acordo com os valores da função objetivo dos indivíduos. A operação de recombinação combina valores dos pais para gerar descendentes usando uma função aritmética. O operador de mutação substitui o valor de uma entrada (gen) por outro valor selecionado aleatoriamente.

Zhan e Clark

Zhan e Clark descrevem um método para a geração automática de dados de teste usando um algoritmo de recozimento simulado para modelos Matlab/Simulink (Zhan e Clark, 2005). Simulink é um software para modelagem, simulação e análise de sistemas dinâmicos. Recursos de simulação permitem que o modelo seja executado e observado.

Modelos são compostos de blocos conectados por linhas. Cada bloco recebe entradas, implementa alguma função, e produz resultados como saída. As saídas de blocos formam

entradas para outros blocos, o que é representado por linhas conectando saídas e entradas. Alguns blocos formam ramos: “for”, “if”, “switch”, “while”, dentre outros. Blocos tipo switch (os únicos tratados no trabalho) possuem três entradas. A segunda entrada controla se a primeira ou se a terceira entrada será considerada para produzir a saída. Este controle é feito pela avaliação ($V_p \geq \text{limiar}$), sendo que V_p é o valor da segunda entrada (de controle) e limiar é um parâmetro associado ao switch.

São introduzidos defeitos no modelo em teste por meio de operadores de mutação. Esses defeitos provocam perturbação nos valores pela adição, multiplicação, ou atribuição de valores presentes nas entradas dos blocos. São gerados dados que matam os mutantes, isto é, que fazem com que a saída final do modelo mutante seja diferente da saída do modelo original. A avaliação de quão bem um dado de entrada satisfaz um requisito de teste é baseado em o quão distante o defeito injetado se propaga no modelo mutante em relação ao original, para qualquer caminho entre o defeito introduzido e a saída do modelo.

A função objetivo é calculada pela comparação dos estados de execução do modelo original e do modelo mutante. Para que os dois modelos apresentem saídas distintas é necessário que duas condições sejam satisfeitas: 1) os valores de sinal após o ponto onde o defeito foi inserido sejam diferentes; 2) a diferença de valores atinja a saída do modelo. A condição 1 é tratada de forma análoga ao nível de aproximação de outros trabalhos (ex: Zhao et. al.). A condição 2 é abordada pelo rastreamento de pontos ao longo da estrutura do modelo e busca fazer com que cada ponto apresente diferenças de valores entre os dois modelos. A função objetivo considera valores detectados em condições de ramos nos comandos do tipo switch por meio da inserção de pontos de prova. São computados custos relacionados às diferenças de valores entre o modelo e o mutante no comando para compor o valor objetivo a ser minimizado para obtenção do dado de teste que mate o mutante.

Ghani et al.

Em um trabalho posterior Ghani et al. comparam o desempenho de algoritmos de busca para exercitar modelos Matlab/Simulink (Ghani et al., 2009). Além de considerar blocos switch, são tratados também blocos de operadores lógicos e blocos de operadores

relacionais. Operadores lógicos recebem parâmetros numéricos e um parâmetro operador lógico (“and”, “or”, “not”, “xor”, etc.). Operadores relacionais recebem parâmetros numéricos e um parâmetro operador relacional (“>”, “<”, “==”, “≤”, etc.).

Um Algoritmo Genético baseado em um toolbox da empresa MathWorks (<http://www.mathworks.com/>) é comparado a um algoritmo de Recozimento Simulado para a geração de dados de teste. Dois experimentos foram realizados. No primeiro experimento os algoritmos foram aplicados considerando o critério Cobertura-de-Todos-os-Caminos, sendo que caminhos representam uma combinação de blocos tipo switch. Satisfazer este critério estrutural requer que o conjunto de testes exercite todas as possíveis combinações de predicados tipo switch. O segundo experimento foi considerado o critério Cobertura-de-Ramos para blocos Simulink dos tipos: relacional, condicional e switch. Este critério requer que cada avaliação de condição seja executada pelo menos uma vez.

Blanco et al.

Blanco et al. apresentam uma abordagem para geração de dados de teste para serviços web especificados em BPEL – *Business Process Execution Language*. (Blanco et al., 2009).

Especificações *BPEL* representam o comportamento de processos de negócio baseados em composições de serviços web. A especificação dos processos de negócio consiste de conjuntos de atividades que podem ser executadas e que podem ser básicas ou estruturadas. Atividades básicas podem invocar serviços web, receber invocações, ou podem atualizar valores de variáveis. Atividades estruturadas podem estabelecer uma ordem seqüencial de atividades; podem realizar a repetição da execução de um conjunto de atividades até que uma condição seja satisfeita; ou ainda, podem estabelecer fluxos concorrentes de atividades.

A técnica de busca utilizada é a Busca Dispersa – BD (Scatter Search), método evolutivo que trabalha com uma população de soluções para o problema a ser resolvido (Glover, Laguna e Martí, 2000).

O objetivo do sistema é gerar dados de teste que façam com que cada transição dos processos de negócio seja exercitada. Este objetivo geral é dividido em sub-objetivos, cada

um consiste em gerar dados que alcancem uma transição particular T_k do grafo de estados. Durante o processo de geração de dados, informações são associadas a cada transição. Essas informações são chamadas Conjuntos Referência S_k , associados às transições T_k do grafo. Cada conjunto referência S_k mantém as seguintes informações: dado de teste que alcança a transição T_k ; caminho coberto pelo dado de teste até T_k ; distância para que o dado de teste execute a “transição irmã” de T_k ; e distância da próxima transição a ser exercitada.

O algoritmo (chamado TCSS-LS) inicia usando um método de geração de diversidade para gerar P soluções diversas. O Conjunto Referência é gerado com as melhores soluções de P e também as mais diversas em relação às soluções já existentes no Conjunto Referência. São produzidos subconjuntos do Conjunto Referência e é feita a combinação de soluções para obter novas soluções, às quais o método de melhoria é aplicado. Estas novas soluções são consideradas para a atualização do Conjunto Referência. Soluções melhores podem ser incluídas no Conjunto Referência, enquanto soluções piores neste conjunto são removidas. O modelo é executado com cada solução e o conjunto de transições alcançadas é atualizado. Em cada iteração o TCSS-LS seleciona uma transição alvo e gera o conjunto de soluções para o Conjunto Referência da transição. A solução final fornecida são os casos de teste que exercitam cada transição do modelo.

Tracey et al.

Tracey et al. trabalham em uma proposta de desenvolver um framework genérico para a geração de dados de teste baseada em otimização, permitindo a automação da geração para critérios de teste estrutural, para critérios de teste funcional e também para o teste de propriedades não funcionais (Tracey et al., 1998).

O teste de falhas em relação a uma especificação é tratado com o uso de uma especificação formal com o uso de notação SPARK-Ada, que define expressões para prova que consistem de pré-condições e de pós-condições para um subprograma. Variáveis nas pré-condições referem-se a valores de entrada (antes da execução) e variáveis na pós-condição referem-se a valores de saída ou valores de entrada. Para mostrar que uma

implementação não satisfaz totalmente a sua especificação é necessário encontrar um caso de teste que, antes da execução do código, satisfaz a pré-condição e, após a execução, não satisfaz a pós-condição.

A geração de dados segue a abordagem dinâmica e aplica algoritmos de busca para procurar soluções que satisfaçam a pré-condição antes da execução do código e satisfação a negação da pós-condição após a execução (o que é equivalente a não satisfazer a pós-condição).

A Função objetivo é calculada da seguinte forma: inicialmente a pré-condição e a negação da pós-condição são convertidas para a forma disjuntiva normal (DNF). A solução para qualquer disjunção representa, portanto uma solução para toda a condição. Pares de pré-condição e pós-condição (na forma DNF) são formados em uma conjunção. Cada um desses pares representa uma das possíveis maneiras em que o software poderá falhar. O processo de busca tenta encontrar a solução para cada um desses pares, um por vez.

O algoritmo de busca empregado é o Recozimento Simulado e a função objetivo busca minimizar o erro que faz com que os pares pré-condição e pós-condição não sejam satisfeitos. Trata-se da função de ramo definida por Korel e usada também em outros trabalhos (Michel et. al, Bueno, etc).

Os autores abordam também o teste de condições de exceção. No teste dessas condições o foco é sobre situações em que a condição acontece, por exemplo, a ocorrência de um problema de *overflow* numérico. As condições que causam a exceção representam as restrições que devem ser satisfeitas, em um ponto particular durante a execução do programa, para que a exceção ocorra. A abordagem para a geração de dados e a função objetivo são análogas às empregadas para o teste de falhas em relação à especificação, descritas anteriormente. No entanto, os valores das variáveis considerados devem ser aqueles obtidos no ponto exato no código onde a condição de exceção aparece.

Apêndice C – Detalhamento da Base Conceitual da Técnica DOTG

Diversidade e Software: Aumentando a Confiabilidade de Sistemas por Meio da Tolerância a Defeitos

As abordagens que aplicam conceitos de diversidade a questões de software são consideravelmente mais próximos e relevantes para a técnica de teste proposta neste trabalho. A seguir é feita uma descrição da aplicação de conceitos de diversidade à área de Tolerância a Defeitos, aplicação já amplamente explorada e com resultados práticos alcançados.

Não se trata de uma descrição exaustiva, apresenta-se uma seleção de idéias relevantes no contexto deste trabalho. Além disso, as seções seguintes destacam mais os conceitos envolvidos, as técnicas e algoritmos, e menos os procedimentos de avaliação e os resultados obtidos com as abordagens.

O Conceito de Tolerância a Defeitos

Uma aplicação amplamente conhecida do conceito de diversidade em software encontra-se na área de confiabilidade, mas precisamente na abordagem denominada Tolerância a Defeitos (*fault tolerance*)¹⁷. Esta abordagem visa garantir a continuidade de um determinado serviço, mesmo na presença de defeitos que causem estados de erro na execução do software. Ou seja, sistemas tolerantes a defeitos continuam a fornecer as saídas esperadas, no tempo esperado, mesmo quando defeitos existentes (não descobertos previamente) são ativados e provocam erros nos estados internos de execução do software ex: (Randell, 1975), (Avizienis, 1985), (Murugesan, 1989).

¹⁷ Trata-se de uma área comumente designada como “tolerância a falhas”. Utiliza-se a tradução do inglês (tolerância a defeitos) para manter a uniformidade das definições utilizadas.

Trata-se de um conceito antigo, descrito por Babbage em 1824, referindo-se a sua máquina (mecânica) de cálculo: “a maneira mais eficaz de checar erros que acontecem no processo de computação é fazer com que a mesma computação seja feita por máquinas separadas e independentes (...) e esta checagem é ainda mais decisiva se as máquinas fazem as suas computações por métodos diferentes” (citado em Avizienis et al., 2004). Littlewood et al. ilustram que a lógica da Tolerância a Defeitos é a mesma da antiga crença de que “duas cabeças pensam melhor do que uma” (Littlewood et al., 2001).

O ingrediente chave para a Tolerância a Defeitos é a diversidade. A diversidade de projeto e a diversidade de dados, explicadas a seguir, são duas abordagens bem estabelecidas e que buscam fazer com que a chance de resultados incorretos de um sistema seja reduzida, na medida em que se aumenta o nível de diversidade presente no sistema.

Cabe observar que o papel da diversidade em Tolerância a Defeitos é a de diminuir a chance de que falhas ocorram durante a utilização do software. O objetivo do teste de software, ao contrário, é provocar falhas durante a execução do software; isto é, o papel que se deseja da diversidade no teste é justamente o oposto deste papel na Tolerância a Defeitos: deseja-se aumentar a chance de que falhas ocorram durante o teste. O paradoxo é apenas aparente, embora as duas áreas – Tolerância a Defeitos e Teste – utilizem o conceito de diversidade como ferramenta, a primeira área aplica a diversidade para estabelecer componentes redundantes visando evitar falhas, como será descrito a seguir. Já a segunda área utiliza a diversidade para estabelecer dados de teste minimamente redundantes visando provocar falhas.

Diversidade de Projeto

Uma abordagem para a tolerância a defeitos é o “Software com N-versões” (*N-Version Software - NVS*) - ou “Programação N-versões” (*N-Version Programming*) - na qual múltiplos programas funcionalmente equivalentes (as versões) são gerados de forma independente a partir da mesma especificação inicial (Avizienis, 1985).

Estas versões são todas executadas conjuntamente (de forma concorrente, paralela ou sequencial) para cada dado de entrada. A partir dos resultados produzidos (dados de saída), alguma saída de consenso, ou a melhor saída, é definida em tempo de execução por um “gerador de consenso”, referido também como árbitro. Este consenso pode ser feito tomando-se a maioria das saídas, calculando-se a média das saídas, ou por algum outro mecanismo. Portanto, a geração do consenso mascara resultados incorretos das versões com defeitos e gera uma saída correta¹⁸. Isto é possível desde que a maioria das versões (no caso de consenso por maioria) produza o resultado correto (Knight e Leveson, 1986).

No caso (extremo) de todas as versões serem idênticas, isto é, serem apenas cópias feitas de um único software, a abordagem obviamente será ineficaz. Neste caso quando um dado de entrada provocar a falha, todas as versões falharão e o resultado final será incorreto¹⁹. A diversidade entre as versões, nula neste exemplo, claramente interfere na eficácia da abordagem.

Diversidade de Projeto Alcançada por Meio da Diversidade de Processos

Para que a abordagem *Software com N-Versões* seja eficaz e resulte realmente em uma maior confiabilidade do sistema, é necessário promover a diversidade entre as várias versões. As versões devem ser desenvolvidas por diferentes programadores, usando diferentes técnicas, metodologias, algoritmos, linguagens de programação e compiladores. A comunicação entre os programadores pode ser controlada, para impedir alguma possível “colaboração” e aspectos como algoritmos, estilo de programação e linguagem, podem ser impostos visando “forçar” a diversidade dentre as versões tanto quanto possível (Lyu e He, 1993).

A lógica é que, por meio de processos diversos (diversidade nas técnicas, práticas e pessoas), criam-se versões diferentes umas das outras e; em especial, versões nas quais seja pouco provável que os defeitos existentes em uma versão sejam idênticos aos presentes em

¹⁸ Notar que as versões que produziram a saída correta também podem possuir defeitos, mas que não foram ativados da execução em questão.

¹⁹ No caso de redundância de hardware esta afirmação não se aplica. Hardware pode falhar devido ao desgaste. Isto que faz com que o uso de versões idênticas de hardware faça sentido.

outras versões. O que se espera ao se utilizar a técnica Software com N-Versões é que essas diferenças entre as versões diminuam a chance de que, se uma versão falhar para um dado de entrada particular, outra versão também irá falhar. Esta diversidade de comportamento de falha é na verdade o objetivo da diversidade de projeto.

Pontos de Vista Para Quantificar a Diversidade de Projeto ao Longo do Desenvolvimento

Esforços para criar métricas de diversidade de software foram realizados. O objetivo foi de criar uma visão menos simplista da diversidade. Lyu e outros definem diferentes pontos de vista para caracterizar e quantificar a diversidade na abordagem Software com N-Versões (Lyu et al., 1992).

A diversidade é avaliada em termos de distintas características:

1. Diferença estrutural entre as versões: medidas de complexidade e tamanho das versões, tais como: número de linhas de código, Ciência de Software de Halstead (números de operadores e operandos); número de decisões; e complexidade de McCabe;
2. Diferenças entre os defeitos encontrados nas versões: considerando os diferentes programadores, em um determinado ciclo de desenvolvimento, são registrados os defeitos encontrados. A proporção de defeitos diferentes encontrados em relação aos defeitos comuns das versões é uma medida de diversidade de defeitos;
3. Diferenças na propensão a defeitos (fault-proneness) entre as versões; chamada de “though-spot diversity” esta medida visa caracterizar o nível de dificuldade encontrado pelos programadores para construir as versões. É computado o número de defeitos encontrados em diferentes “partes” das versões²⁰. A diversidade na distribuição dos defeitos nas partes em cada versão (associadas a

²⁰ O sentido de “partes” não é definido precisamente. Uma parte é um segmento de código, associados a alguma parte da especificação do software.

diferentes programadores) sugere dificuldades de natureza distinta entre os programadores;

4. Diferença no comportamento de falha entre as versões. A diversidade de falhas busca avaliar a frequência com que duas ou mais versões falham para um mesmo dado de entrada, fato referido como “falhas idênticas”. É computada a proporção de falhas idênticas em relação ao total de falhas.

As medidas definidas foram aplicadas em seis versões de um software de controle de vôo; cada versão codificada em uma linguagem diferente. Os autores destacam fatos observados, referentes às características descritas anteriormente:

1. Algumas medidas de complexidade estrutural variaram entre as versões, outras mostraram pouca variação. Houve um relacionamento expressivo entre o número de operadores e operandos da versão e o número de defeitos encontrados na versão. Este mesmo fato ocorreu para a medida de complexidade ciclomática das versões;
2. Dos 92 defeitos encontrados apenas 2 foram coincidentes em diferentes versões;
3. Foi analisado o percentual de defeitos em cada componente (funções e procedimentos) que formam as versões. Algum nível de diversidade foi encontrado, no entanto, em 3 das 6 versões a maioria dos defeitos situavam-se em um mesmo componente da versão;
4. Em relação à diversidade de falhas, das possíveis 20 configurações de 3 versões, 4 apresentavam probabilidades de falha (conjunta) maiores que zero. Considerando as 6 possíveis configurações de 5 versões, 3 também possuem probabilidades de falha (conjunta) maiores que zero. Esse resultado (probabilidade de falha maior do que 0) ocorre porque, considerando as 6 versões, foram identificadas no total 30 situações de falhas idênticas.

Neste trabalho um estudo de caso foi conduzido com apenas um sistema com 6 versões, o que faz com que os resultados sejam específicos e pouco generalizáveis.

Também não é abordada a relação de cada uma das características de diversidade e a confiabilidade final do sistema N-Versões. Apesar disto, o trabalho é uma contribuição no sentido de a abordagem Diversidade de Projeto utilizar proativamente o conceito de diversidade, ao invés de esperar que a diversidade no comportamento de falhas surja “espontaneamente” do fato de diferentes programadores construírem as diferentes versões. A proposta é que a diversidade das diferentes versões seja avaliada sob diferentes perspectivas e seja “induzida” ao longo do desenvolvimento, buscando minimizar assim a chance de falhas no sistema.

Diversidade de Projeto Quantificada a Partir do Domínio de Entrada do Software

Saglietti (1990) sugere que a medida de diversidade pela quantificação da dissimilaridade do particionamento do domínio de entrada das diferentes versões. É observado que cada programa induz um particionamento do domínio de entrada em subconjuntos disjuntos de dados de entrada – ou partições. Cada partição é associada a um diferente caminho no grafo de fluxo de controle do programa.

A abordagem proposta consiste essencialmente em comparar o particionamento do domínio de entrada das diversas versões. A idéia é que no caso de um particionamento idêntico do domínio de entrada é provável que um raciocínio idêntico tenha sido usado no desenvolvimento das versões, levando assim a uma maior chance de defeitos coincidentes. Por outro lado, quando as versões induzem a um particionamento diferente do domínio de entrada, um erro comum, associado a um determinado dado de entrada, pode propagar-se em “direções diferentes” (ou caminhos diferentes) nas versões distintas. Isto pode ocorrer caso o dado de entrada que provoca o erro esteja situado em uma interseção entre dois diferentes subdomínios das diferentes versões.

A avaliação do nível de diferença no particionamento do domínio de entrada de diferentes versões não é de fácil obtenção por meio de análise estática. Os autores sugerem analisar o comportamento dinâmico do programa para avaliar esta diferença. Durante o teste aleatório, os dados de entrada são executados incrementalmente. A cada novo dado de entrada executado a cobertura atingida nas várias versões (cobertura de caminhos) é

medida. O desvio entre as curvas de cobertura acumulada das versões é interpretado como uma medida de diversidade de projeto entre as versões.

Trata-se de uma abordagem mais concreta para medir a diversidade das versões, baseada em suas propriedades estruturais, avaliadas dinamicamente durante o teste. No entanto, diferentes conjuntos de teste podem levar a valores distintos de diversidade; os autores sugerem que sejam computados valores médios de várias execuções. Outro ponto importante é que o incremento de cobertura nas versões pode ser quantitativamente igual (sugerindo ausência de diversidade), mas ser associado à cobertura adicional de diferentes caminhos. Neste caso a diversidade existente (o mesmo dado de entrada executa caminhos distintos) não é capturada pela técnica.

Diversidade de Projeto Avaliada por Meio de Inserção de Defeitos e Análise de Falhas

Outro trabalho com abordagem semelhante foi desenvolvido por (Chen, et al., 2001). Neste trabalho, perturbações em fluxos de dados e em constantes simulam a introdução de defeitos ou erros nos programas. O comportamento de falha é avaliado quantitativamente para estimar a diversidade do software de múltiplas versões.

As perturbações nos fluxos de dados são feitas por um “gerador de anomalias” inserido nos softwares originais. Por exemplo, considerando uma variável x a perturbação pode ser feita por meio dos comandos $x = x + \alpha$ ou $x = x\alpha$, sendo α o parâmetro de perturbação. As mesmas perturbações são inseridas nas duas versões a serem avaliadas e o software original é usado como uma “gold version” para a comparação de saídas produzidas e avaliação da ocorrência da falhas.

Têm-se então processos de busca realizados nos domínios de entrada das duas versões do software, um processo distinto para cada versão. O objetivo é avaliar os efeitos no software das perturbações inseridas. A busca é feita em duas etapas, com diferentes objetivos: etapa 1 – busca para identificar um ponto do domínio de entrada no qual o software falhe; e etapa 2 – busca realizada a partir deste ponto para estabelecer o tamanho e os limites da região de falha. Após descoberto certo ponto de falha, um algoritmo de busca

do tipo “branch and bound” é utilizado para buscar as regiões contínuas na vizinhança do ponto.

A partir das regiões de falha das duas versões do software é possível calcular a diversidade dessas versões. A diversidade, que define a chance de ocorrer uma falha conjunta das duas versões, está associada a como as regiões de falha se sobrepõem e pode variar de zero a um. No primeiro caso (diversidade = 0) não existe melhoria no uso de duas versões em relação ao uso de apenas uma versão, pois a área de falha coincidente é igual à menor das regiões de falha, dentre as duas versões. No segundo caso (diversidade = 1) as duas versões não possuem região de falhas coincidente, o que seria uma situação ideal.

A premissa desta abordagem é que se duas versões do software mostram uma alta diversidade para perturbações de dados artificiais, provavelmente as duas versões também terão uma alta diversidade para defeitos reais, independentemente de onde eles estejam nas estruturas do software.

Diversidade de Projetos: Sumário

Essencialmente as várias abordagens associadas à diversidade de projeto buscam utilizar algum tipo de redundância de software e mecanismos de escolha de saídas; busca-se tolerar a existência de defeitos – ou erros – e manter o desempenho esperado do sistema mesmo na presença desses defeitos em algumas das versões.

Nem todas as técnicas existentes foram descritas na seção anterior. Por exemplo, a técnica Bloco de Recuperação (Recovery Block) também utiliza múltiplas versões do software; no entanto, nesta técnica o objetivo é detectar falhas enquanto o sistema está em operação e recuperar o sistema para o último estado livre de falha (Littlewood et al., 2001). Após esta recuperação outra versão do software, com mesma função, é executada. Para que as falhas sejam identificadas é necessário que o módulo em execução passe por “testes de aceitação”; caso o teste indique uma falha, o sistema recupera estados (roll back) e inicia a execução de um módulo (versão) alternativo a partir de um ponto intermediário, ou estado do sistema – o Ponto de Recuperação. Esta operação ocorre até que um resultado aceitável seja obtido, ou que todos os módulos sejam executados sem sucesso.

Na diversidade de projeto, em suas várias perspectivas, a variabilidade tem um papel essencial: a variabilidade de equipes, de linguagens de programação, de compiladores, etc., busca na verdade resultar em versões estruturalmente diferentes. Em última análise espera-se que as versões apresentem variabilidade no comportamento de falha e conseqüentemente se ganhe em confiabilidade do sistema em uso.

Apesar da tolerância a defeitos e o teste serem áreas distintas, e possuírem objetivos diferentes, o processo que levam à ocorrência das falhas são os mesmos durante o teste (neste caso desejáveis e consonantes com o objetivo do teste) e durante a utilização de um software (indesejáveis e que levam a menores expectativas de confiabilidade). A diversidade de projeto, naturalmente, oferece uma abordagem para lidar com falhas do software em uso e não falhas durante o teste. A abordagem diversidade de dados tem este mesmo objetivo, mas oferece uma alternativa ao uso de várias versões de software. Esta abordagem considera a natureza do processo de ocorrência de falhas como base para tolerar defeitos.

Diversidade de Dados

A Diversidade de Dados é outra abordagem para que certas classes de defeitos de software sejam toleradas. Comumente as falhas de software ocorrem em alguns casos especiais, enquanto que para a maioria das situações o software funciona de forma correta. Este fato é a base para a definição da abordagem Diversidade de Dados, que pode ser vista como ortogonal à Diversidade de Projeto, apresentada na seção anterior.

Uma curiosa aplicação deste conceito remonta ao século XVIII. Tabelas astronômicas lunares, publicadas no *Nautical Almanac* entre 1765 e 1809, eram fundamentais para a orientação geográfica dos navegantes nesta época (Croarken, 2003). Para computar estas tabelas era atribuída a um computador (humano) a tarefa de realizar medições e calcular a posição da lua (latitude e longitude) ao meio dia, durante todos os dias da semana. Outra pessoa, chamada de anti-computador, devia realizar as mesmas medições e cálculos, mas considerando a posição da lua à meia noite. Os resultados obtidos pelo computador e pelo

anti-computador eram revistos por eles e então enviados a uma terceira pessoa, chamada de comparador e responsável por checar e combinar os resultados obtidos. Pela natureza do problema (simetria de posições da lua de dia e à noite) e pela maneira com que o cálculo era realizado, tornava-se possível ao comparador identificar as discrepâncias existentes entre os cálculos e determinar qual informação deveria ser publicada nas tabelas.

Nesta aplicação, diferentes “hardwares humanos” (computar e anti-computador) seguiam um mesmo procedimento para realizar as observações e efetuavam os mesmos cálculos. Apesar de cada um utilizar dados diferentes (posição da lua de dia e a noite) eles deviam chegar a resultados semelhantes, que podiam assim ser comparados para avaliação de correteude.

Segundo Ammann e Knight “se o software falha sob certo conjunto de condições de execução, uma pequena mudança nessas condições pode permitir que o software produza resultados aceitáveis” (Ammann e Knight, 1988). Esta mudança nas condições de execução pode ser conseguida por meio de alterações nos dados, que podem ocorrer naturalmente no tempo (separação temporal); ou pela alteração deliberada dos dados (re-expressão de dados). A separação temporal baseia-se na observação de que certos defeitos que provocam falhas em uma execução, nem sempre causam a falha se as mesmas entradas forem submetidas em uma segunda execução. O sistema funciona corretamente na segunda execução pela chance de reordenar eventos assíncronos²¹. Usuários que reiniciam seus computadores quando há ausência de resposta do sistema utilizam, de maneira intuitiva, a diversidade por separação temporal para solução dos seus problemas.

Diversidade de Dados Para Tolerância a Defeitos de Software Usando Re-expressão de Dados

A abordagem de Ammann e Knight é do tipo re-expressão de dados. Um sistema com diversidade de dados produz, utilizando uma fórmula de re-expressão, um conjunto de

²¹ Tais tipos de defeitos (bugs) foram chamados de “Heisenbugs” em alusão ao aparente não determinismo e incerteza de manifestação dos defeitos. Heisenbugs são de difícil identificação e podem estar relacionados a fatores como: compilador e modo de compilação usados (modo “debug”), execução concorrente com situações específicas e incorretas de sincronismo entre processos; variáveis não iniciadas, dentre outros.

dados de entrada relacionados e executa o mesmo software com cada um desses dados. Um algoritmo de decisão (ou gerador de consenso) determina a saída do sistema. A eficácia desta abordagem depende da habilidade do algoritmo de re-expressão em produzir dados de entrada situados fora da região de falha²² a partir de um dado de entrada situado dentro desta região.

A implementação da diversidade de dados pode ser feita por meio de “Sistemas N-Cópias” (N-copy system). Nesta abordagem N cópias idênticas do programa são executadas em paralelo, cada uma com um dado de entrada distinto produzido por re-expressão. O sistema seleciona a saída a ser usada por meio de um algoritmo de decisão (também chamado de “voting scheme”).

A re-expressão de dados gera dados de entrada que sejam equivalentes ou próximos uns dos outros no aspecto lógico. Para um dado de entrada x , submetido à execução, o programa produz uma saída $P(x)$. O algoritmo de re-expressão R transforma o dado de entrada original x em um dado de entrada y , $y = R(x)$; quando $P(y)$ é um resultado correto e $P(x)$ não o é, o resultado da execução $P(y)$ pode ser usado para tolerar o defeito revelado quando x é utilizado como entrada para a execução (resultando em $P(x)$). Um algoritmo de re-expressão pode ser aplicado diversas vezes e gerar vários diferentes dados de entrada y (y_1, y_2, \dots, y_n) a serem usados para executar P . A Figura 37 (original de Amman) ilustra a re-expressão de dados de entrada.

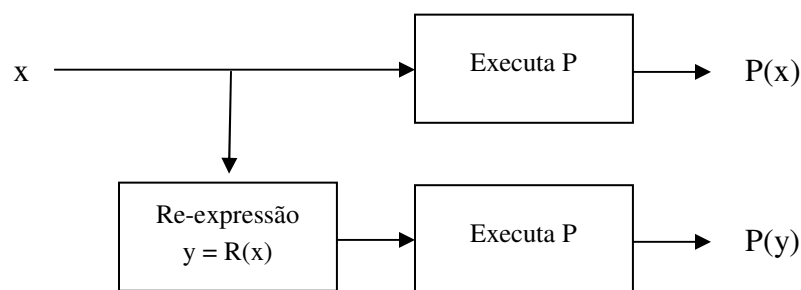


Figura 37. Re-expressão de dados de entrada na diversidade de dados

²² A região de falha define o conjunto de pontos do domínio de entrada (dados de entrada) que provocam a falha do software.

Os algoritmos de re-expressão dependem de manipulação numérica, embora os autores destaquem que outras formas de transformações – não numéricas – são possíveis. Exemplo de re-expressão numérica: para variáveis reais um algoritmo altera o valor da variável por um pequeno percentual; outro exemplo: se um programa manipula pontos de entrada cartesianos e apenas a posição relativa dos pontos importa, um algoritmo de re-expressão pode fazer uma translação do sistema de coordenadas para uma nova origem.

O novo dado de entrada y (gerado por uma re-expressão) pode conter exatamente a mesma informação de x , ou pode conter uma informação diferente, mas próxima de x . No primeiro caso, y faz parte do “conjunto de saídas idênticas” (I), pois y produz a mesma saída do que x produziria caso o programa estivesse correto. Apesar de desejável tal situação não é a mais comum. O conjunto de saídas idênticas é um sub-conjunto do “conjunto de saídas válidas” (V). Neste segundo caso, y não contém informação idêntica a de x e $P(y)$ produz um resultado válido (correto), mas diferente do valor produzido com $P(x)$ caso o programa estivesse correto.

Algoritmos de re-expressão que produzem dados de entrada pertencentes ao conjunto I são chamados de exatos, enquanto que os que produzem elementos em V são chamados de aproximados. Se o algoritmo de re-expressão é exato, todas as cópias devem gerar saídas idênticas e, portanto, o algoritmo de decisão por maioria pode ser empregado de forma direta. No entanto, se o algoritmo de re-expressão é aproximado, as cópias podem gerar diferentes resultados válidos (corretos em relação a especificação); nestes casos é necessário um algoritmo de decisão mais complexo para determinar a saída final do sistema.

Deve-se notar que neste caso (algoritmo de re-expressão é aproximado) quando x é fornecido como entrada o resultado correto $P(x)$ é o realmente desejado como resposta do sistema. O algoritmo de decisão neste caso deve, a partir do resultado produzido para cada diferente dado de entrada $(x, y_1, y_2, \dots, y_n)$, determinar a saída do sistema. Segundo os autores esta determinação pode ser feita por alguma forma de consenso. Algum ajuste (A) também pode ser requerido na saída do programa para desfazer a distorção causada pela re-expressão do dado de entrada. Por exemplo, uma translação do sistema de coordenadas,

reversa à translação utilizada dos dados de entrada, pode ser aplicada nos dados de saída. Tem-se assim um “ajuste de pos-execução” que reverte a alteração efetuada nos dados de entrada pelo algoritmo de re-expressão.

Além do modelo descrito anteriormente (Figura 37), os autores também definiram outro modelo usando a diversidade de dados. No Bloco de Re-tentativa (Retry Block) o dado de entrada é executado e a saída produzida é submetida a um teste de aceitação. Se o teste passa, a saída é fornecida como resultado; se o teste falha o algoritmo é novamente executado após a re-expressão do dado de entrada. Este processo é repetido até que o sistema produza uma saída satisfatória, ou até que um limite de execuções seja atingido. Este modelo é análogo ao Bloco de Recuperação (Recovery Block) da abordagem diversidade de projeto.

As abordagens foram avaliadas experimentalmente de forma preliminar em um software de controle de lançamento aéreo com alguns defeitos conhecidos. A eficácia da diversidade de dados foi avaliada em relação a dois parâmetros: o raio do deslocamento usado pelo algoritmo de re-expressão²³ e o efeito deste algoritmo para diferentes defeitos. Ambas as abordagens (Bloco de Re-tentativa e Sistemas N-Cópias) apresentaram resultados bastante próximos.

Em ambas as abordagens maiores deslocamentos feitos na re-expressão tornam o dado gerado por re-expressão com menor chance de causar falha, um resultado esperado. A eficácia, no entanto, varia de forma significativa para os diferentes defeitos. Para alguns defeitos a re-expressão não tem nenhum sucesso em gerar pontos fora da região de falha. Nestes casos a chance do dado gerado por re-expressão falhar é idêntico ao do dado original (o multiplicador de chance de falha, valor calculado pelos autores, é igual a 1 nesses casos). Em outras situações o multiplicador assume valores entre 0 e 1, significando uma eficácia parcial. Ainda, para alguns defeitos este multiplicador de chance de falha assume valor 0, mostrando que nestes casos o algoritmo parece garantir que o dado re-

²³ O algoritmo de re-expressão utilizado move um ponto (x,y) para uma localização aleatória na circunferência centrada em (x,y) e com um raio definido e fixo.

expresso esteja fora da região de falha. No geral dos 7 defeitos analisados 4 defeitos foram tolerados completamente e 3 defeitos não foram tolerados.

Diversidade de Dados Aplicada à Tolerância a Defeitos de Hardware

Uma abordagem próxima à de Ammann e Knight foi proposta em (Oh et al., 2002), denominada “Detecção de erros por dados diversos e instruções duplicadas” (*ED4I – Error detection by diverse data and duplicated Instructions*). O foco deste trabalho, está tolerância a defeitos de hardware ao invés de defeitos de software. No entanto, esta tolerância é implementada por meio de técnicas de redundância de software. A abordagem é proposta como uma alternativa às técnicas de tolerância a defeitos baseadas em redundância de hardware.

Por meio da execução de dois diferentes programas que possuem a mesma funcionalidade, mas utilizam diferentes dados de entrada, é possível detectar defeitos temporários e permanentes. “Defeitos temporários” ocorrem quando o estado de um programa é corrompido por falhas transientes e mudanças indesejáveis de memória (bit-flips). Defeitos permanentes são resultados, por exemplo, de problemas de montagem do hardware que podem levar a *Stuck-at faults*, fazendo com que o valor de algum dado fique fixo em 0 ou 1.

A idéia essencial é que os defeitos provocarão comportamentos distintos na execução dos dois programas. Um resultado incorreto (resultado produzido por um programa corrompido) pode ser detectado ao ser comparado com o resultado produzido pelo outro programa (o que não teve os dados corrompidos). Ainda que o defeito corrompa os dois programas é possível detectá-lo se cada um deles produzir resultados incorretos distintos.

O programa com diversidade de dados é gerado do seguinte modo: o programa P (original) é transformado em um novo programa P’ (modificado). São realizados dois tipos de transformação: transformação de expressões e transformação de condições de ramos.

Na transformação de expressões toda expressão em P é transformada em uma nova expressão em P’ na qual cada variável e cada constante é um k-múltiplo da variável ou

constante em P. Se x é k vezes maior que y diz-se que x é k -múltiplo de y ; k é chamado de fator de diversidade. Por exemplo, para $k = -2$, a expressão $z = x + i * y$ de P é transformada em $z = x + i * y / (-2)$ em P'.

Como os valores para as variáveis em P' são diferentes dos valores originais, é necessário que as inequações presentes nos comandos condicionais sejam também alteradas. Por exemplo, para $k = -2$ o comando `if (i < 5)` em P precisa ser alterado para `if (i > -10)` em P'. A transformação das condições de ramos ajusta as inequações, presentes nos comandos de condição de P', de modo que os fluxos de controle de P e P' sejam idênticos.

Deve-se notar que duas condições devem ser respeitadas pelas transformações: 1) o Grafo de Fluxo de Controle (GFC) de P' é isomórfico em relação a P; e, 2) os valores para as variáveis em P no final de cada bloco de comandos (um nó do GFC), ao serem multiplicados por k , devem ser iguais aos valores das respectivas variáveis em P'. 3) as saídas produzidas por P', são k -múltiplos das saídas de P, isto é, as saídas de P multiplicadas pelo fator k devem ser iguais às respectivas saídas de P' (assim como ocorre com as variáveis e constantes internas); isto permite que os resultados de P' sejam mapeados de volta e comparados com os resultados do programa original.

As transformações de expressões e de condições são realizadas por meio de um *parser* que transforma as árvores que representam as expressões e condições de P em novas expressões e condições em P'.

Avaliações realizadas pelos autores abordam transformações realizadas em variáveis inteiras e de ponto flutuante; é avaliado também como determinar valores para k que maximizem a diversidade do programa transformado e com isto aumentem as chances dos defeitos serem detectados. Valores para k desejáveis são os que garantem que os dois programas não produzirão a mesma saída errônea (chamada integridade de dados – id) e também que maximizem a chance de que os dois programas produzam diferentes saídas para o mesmo defeito de hardware (chamada de probabilidade de detecção de defeito – pdd).

Notar que a técnica ED4I é específica para a identificação de problemas em hardware e não em software. Assim como na abordagem de re-expressão de dados para em software, a solução proposta busca evitar que defeitos ocasionem resultados incorretos durante o uso do sistema.

Diversidade de Dados: Sumário

As duas abordagens apresentadas de diversidade de dados manipulam de alguma forma as informações tratadas pelo sistema software/hardware. Dados de entrada e dados internos são transformados visando induzir a estados na execução do software que, no caso de sucesso das técnicas, permite escapar de estados de execução para os quais o sistema falha. Em outras palavras, a diversidade dos dados utilizados na execução é uma propriedade relevante para que se evitem as falhas.

A abordagem de teste de software desenvolvida nesta Tese também utiliza o conceito de diversidade de dados, mas com um propósito distinto. Ao invés do objetivo de modificar dados de entrada para aumentar a chance de escapar de regiões de falha e obter um resultado útil do software em uso, o objetivo é de promover a diversidade dos dados de teste a serem utilizados para aumentar a chance de revelar defeitos no software durante o teste – encontrar as regiões de falha, ao invés de evitá-las. Dito de outra forma: busca-se evitar que regiões de falha fiquem “escondidas” de conjuntos de dados de teste pouco diversos. Deseja-se também que os dados de teste sejam tão representativos quanto possível do domínio de entrada do software de forma a levar a um ganho de confiança de que o software satisfaz os seus requisitos.

A Relação Entre a Diversidade no Teste e a Confiabilidade de Software

Esta seção descreve sumariamente conceitos importantes na área de confiabilidade de software, característica de qualidade essencial em sistemas computacionais, sobretudo em sistemas críticos, para os quais um comportamento incorreto pode resultar em graves consequências – como perda de vidas, danos ao meio ambiente, ou grandes perdas

financeiras. A confiabilidade de um software em uso é influenciada, dentre outros fatores, pela forma com que o teste do software foi realizado.

Essencialmente, as características dos dados de teste utilizados – como o número de dados de teste ou a cobertura atingida no código – levam a uma maior ou menor chance de revelar defeitos e de aprimorar a confiabilidade do software por meio da remoção desses defeitos. Além disso, informações sobre o teste podem ser utilizadas para estimar a confiabilidade do software durante o teste e prever a confiabilidade do software em uso. No contexto desta Tese é de especial interesse o primeiro aspecto, isto é, avaliar características dos dados de teste que favorecem o aumento da confiabilidade do software em uso, tendo em mente tirar proveito desta informação para aprimorar a geração de dados de teste. Como será destacado nas seções seguintes propõe-se que a diversidade dos dados de teste é um aspecto que pode influenciar a confiabilidade em uso do software.

Modelos de Confiabilidade de Software

A confiabilidade de um software é definida como “a probabilidade de um sistema de software operar sem falhas por um período de tempo específico e sob condições de operação especificadas.” (IEEE, 1990). Trata-se de uma das características de qualidade de software definidas na norma de qualidade de produtos de software ISO/IEC 9126-1²⁴ (ISO/IEC 9126-1, 2001). (Littlewood e Strigini, 2000), (Lyu, 2007).

Um modelo de confiabilidade de software especifica a natureza do processo aleatório que descreve o comportamento das falhas de um software em relação ao tempo (Lyu, 2007). O princípio básico é a realização de um ajuste de curvas dos dados observados sobre falhas do software no tempo, usando um modelo pré-especificado. Esse modelo pode ser parametrizado por meio de técnicas estatísticas e pode permitir estimativas da confiabilidade atual, ou previsões de confiabilidade futura do software, usando para isto técnicas de extrapolação.

²⁴ O conteúdo desta norma inclui um modelo de qualidade de produtos de software. As principais características de qualidade são: funcionalidade; confiabilidade; usabilidade; eficiência; facilidade de manutenção; e portabilidade. Cada característica é organizada em sub-características e estas em atributos

O Teste de Software para Avaliar a Confiabilidade

Na área de confiabilidade de software é amplamente aceito o conceito de usar o Perfil Operacional do software durante o teste. Um perfil operacional de um sistema é uma caracterização quantitativa de como o sistema será utilizado pelos usuários (Musa et al., 1987). Na maioria das abordagens para estimar confiabilidade, os dados de teste são gerados aleatoriamente, mas respeitando o modelo de uso caracterizado pelo perfil operacional.

O perfil operacional permite que sejam alocados os recursos de teste às funções do software tendo como base o quanto cada função será utilizada, fazendo com que as funções a serem mais utilizadas tenham sido mais testadas. Um aspecto importante é que, caso o perfil operacional reflita de forma fiel o perfil de uso futuro, a avaliação de confiabilidade realizada durante a fase de testes pode indicar a confiabilidade esperada quando o software estiver em uso. Como o perfil operacional caracteriza as várias maneiras pelas quais o software será usado, este perfil indiretamente induz algum nível de diversidade no teste em relação ao teste puramente aleatório. Normalmente o teste aleatório é feito considerando-se uma distribuição uniforme, na qual cada possível entrada para o software possui a mesma chance de ser selecionada (Hamlet, 2005).

Impacto de Aspectos do Teste na Confiabilidade do Software

Whittaker e Voas (2000) discutem sobre as limitações presentes e as perspectivas futuras em confiabilidade de software. As ideias nessa área foram desenvolvidas inicialmente a partir dos conceitos e modelos de confiabilidade de hardware. Estes modelos se baseiam essencialmente nas condições e no tempo em operação do hardware, sem ocorrência de falhas, para avaliar a confiabilidade de um componente de hardware. Modelos de confiabilidade de software herdaram esta abordagem e, inicialmente, consideraram o perfil de uso e o tempo de operação do software para avaliação de confiabilidade. O papel do tempo na confiabilidade do hardware é evidente, visto que há desgaste físico e envelhecimento de equipamentos ao longo do tempo. Com software, no

entanto, não há desgaste físico ou envelhecimento. Os autores questionam essa consideração do tempo por meio da questão: “o quão importante é o papel do tempo para revelar defeitos no software?”. A resposta: “depende”; “mais tempo oferece maior oportunidade para testar mais variações de dados de entrada, mas um maior tempo gasto no teste não é o que caracteriza um teste completo”.

Duas situações opostas em relação ao uso do tempo, destacadas pelos autores, ilustram que a qualidade do teste depende de “o que o testador faz ao longo do tempo disponível”. Situação (1): “horas gastas com o software inativo, ou testando mais e mais as mesmas características”; ou situação (2): “as mesmas horas gastas executando, sem interrupções, uma série de testes severos e minimamente sobrepostos”. Certamente espera-se que na segunda situação os testadores tenham mais chances de revelar defeitos do que na primeira situação. Os autores concluem apontando que, para a modelagem de confiabilidade de software, no que diz respeito ao teste, deve-se considerar o número de dados de teste executados e a diversidade desses dados. Segundo Whittaker e Voas “deve-se garantir que os dados de teste sejam diversos o suficiente para formar um conjunto completo de dados de teste.”

Desenvolvimentos posteriores na área de confiabilidade de software passaram a considerar a cobertura atingida pelo teste no código do software como um indicador de o quão completamente (*thoroughly*) o software foi exercitado durante o teste (Chen et al., 2001), (Malaya et al., 2002), (Crespo et al., 2008). Esses trabalhos enfatizam a necessidade de se avaliar o impacto do uso de informação mais significativa do que o tempo gasto em teste, em avaliações de confiabilidade. Os resultados contrapõem os fatores: eficácia do teste; cobertura atingida no teste, e confiabilidade final do software. Merecem destaque alguns pontos identificados nesses trabalhos:

- Um teste eficaz tende a permitir a revelação de um número maior de defeitos e levar a uma maior confiabilidade final do software (Hamlet, 1994);
- Um teste eficaz tende a permitir um crescimento mais rápido da confiabilidade do software ao longo da realização do teste (Crespo et al., 2008);

- Uma maior cobertura do código leva a uma maior confiabilidade do software e a uma menor taxa de falhas em uso (Frankl e Weiss, 1993), (Horgan et al., 1994), (Wong et al., 1994)
- A correlação entre eficácia do teste e a cobertura de nós (comandos) é maior do que a relação entre eficácia do teste e o tamanho do conjunto de teste (Wong et al., 1994).
- A coleta e análise de informações significativas sobre o teste permitem alcançar estimativas mais precisas da confiabilidade (Crespo et al., 2008).

Em Chen et al (2001) informação sobre cobertura do código, além do tempo, é utilizada para ajustar as taxas de falha obtidas. Essencialmente, neste modelo os casos de teste redundantes (que não exercitam novos elementos requeridos e nem revelam defeitos) são identificados e é aplicada uma “taxa de compressão” para ajustar a estimativa da confiabilidade do software. O objetivo é impedir que a confiabilidade seja sobreestimada pela aplicação de casos de teste redundantes. Chen et al concluem que informações sobre cobertura são mais relacionadas à confiabilidade do que informações sobre o número de casos de teste executados.

Malaiya et al (2002) destacam que a confiabilidade é quantificada durante o teste por meio de execução do código com dados de entrada gerados segundo o perfil de uso do software. No entanto, várias técnicas não aleatórias buscam uma maior eficácia do teste por meio da seleção de dados de entrada com foco em funções ou partes do código mais propensas a terem defeitos. Esta é a motivação para o desenvolvimento de métodos para medir e estimar a confiabilidade de software quando o teste não é aleatório. Neste sentido é proposto um modelo de crescimento logarítmico de confiabilidade baseado no conhecido modelo Musa-Okumoto (Musa et al., 1987). Contudo, no modelo proposto utilizam-se informações sobre a cobertura atingida considerando critérios estruturais de teste – cobertura de nós, ramos, usos computacionais e usos predicativos (Malaya et al., 2002). Segundo os autores há correlação entre a cobertura de elementos requeridos de critérios estruturais e a revelação de defeitos (chamada de cobertura de defeitos). O modelo proposto

prevê um aumento inicialmente lento da cobertura de defeitos até que se atinja um limiar de cobertura, a partir do qual há uma “curva ascendente” que passa a ser aproximadamente linear relacionando a cobertura estrutural do teste com a cobertura de defeitos.

Crespo et al. apresentam um novo modelo binomial de confiabilidade de software baseado em informação de cobertura de critérios de teste estrutural, o “*Binomial Software Reliability Model Based on Coverage – BMBC*” (Crespo et al., 2008). A informação sobre cobertura alcançada no teste é utilizada para aumentar a propriedade preditiva do modelo. Medidas de cobertura avaliam a adequação de um conjunto de dados de teste em relação ao grau com que o código é exercitado. Segundo os autores, “se existe um aumento da cobertura, isto é, um novo dado de teste exercita uma área do código ainda não exercitada, aumentam as chances de revelar novos defeitos”, este é o ponto central da proposta de que informação sobre cobertura fornece uma base concreta para as atividades de teste e para medir de forma apropriada a confiabilidade.

O modelo *BMBC* (Crespo et al., 2008) considera o número de dados de teste executados ao invés do tempo de execução, além do complemento da cobertura atingida pelos dados de teste executados ($1 - \text{cobertura}$), com respeito a critérios estruturais de teste. Neste modelo o comportamento da taxa de ocorrências de falhas do software segue uma distribuição binomial de probabilidades. A utilização do complemento da cobertura atingida é justificada pelo fato de que a taxa de falhas do software é relacionada com o aumento da cobertura de um critério de teste. Ao longo do teste, à medida que a cobertura aumenta, defeitos são revelados e removidos, fazendo com que a taxa de falhas tenda a diminuir. Isto é, a diminuição de $(1 - \text{cobertura})$ é acompanhada da diminuição da taxa de falhas, conseqüente do aumento do número de defeitos removidos. Deve-se notar que a taxa de falhas do software depende da taxa de falhas por defeito e do número de defeitos no software.

A aplicação do *BMBC* foi realizada em um experimento que utilizou um software da agência espacial européia (software *SPACE*) com 33 defeitos reais ²⁵. Dados de teste foram

²⁵ “Defeito real” significa um defeito inserido ao longo do desenvolvimento do software, em oposição a defeitos implantados no software artificialmente para realização de avaliações.

selecionados aleatoriamente segundo o perfil operacional do software. A cada ocorrência de falha, o número de casos de testes executados e a cobertura atingida com os dados de teste aplicados são até então foram registrados. Os critérios utilizados foram: Todos os Nós, Todos os Arcos, Todos os Potenciais Usos e Todos os Du-Caminhos. O significado da cobertura atingida varia dependendo da força (strenght) do critério, aspecto considerado no modelo por meio da normalização de informações sobre cobertura. A confiabilidade estimada pelo modelo proposto foi comparada à confiabilidade real, obtida em uma medição por força bruta, e também comparada à confiabilidade obtida com diversos outros modelos. Os resultados mostram que BMBC permite estimativas mais próximas da confiabilidade real do que os outros modelos avaliados.

Diversidade do Teste e Confiabilidade de Software: Resumo

Os trabalhos descritos anteriormente mostram que um aumento de cobertura do teste, avaliada por meio da aplicação de critérios de teste estrutural, tende efetivamente a levar a uma maior chance de revelar defeitos e, assim, aumentar a confiabilidade do software. Observa-se também que informações sobre cobertura são mais relacionadas à confiabilidade do que informações sobre o número de casos de teste executados. É razoável concluir, portanto que alguma propriedade dos dados de teste – não apenas o número deles – causa o aumento da confiabilidade do software.

Ao se realizar o teste utilizando técnicas baseadas em particionamento (técnicas que subdividem o domínio de entrada em subdomínios, como no teste estrutural; são tratadas na próxima seção) quando se alocam dois ou mais dados de teste para uma mesma partição não ocorre um aumento da cobertura do teste. Considerando que a aplicação de um critério de teste em um programa resulte na divisão do domínio de entrada em P partições e que sejam executados N dados de teste, com $N \leq P$ (tamanho do conjunto de dados de teste menor que, ou igual ao número de partições). Neste caso a cobertura máxima é atingida, somente se cada dado de teste situar-se em um subdomínio distinto; ou seja, se dois ou mais dados de teste forem alocados a um mesmo subdomínio, algum outro subdomínio não será exercitado, fazendo com que a cobertura atingida seja menor do que a máxima possível

(considerando-se o valor de N). Ainda neste caso, a cobertura de 100% das partições é possível apenas se $N = P$ e se cada dado de teste ocupar uma partição distinta. Naturalmente, a cobertura de 100% das partições é possível também se $N > P$ ²⁶. Portanto, para que a cobertura máxima seja atingida nas técnicas baseadas em particionamento, cada dado de teste deve ser diferente o suficiente dos outros dados de teste.

A característica que faz com que um dado de teste seja “bom”, no contexto da confiabilidade do software, é que ele provoque um aumento de cobertura; isto é, cada dado de teste deve exercitar algum elemento requerido ainda não exercitado pelos outros dados de teste. Está implícito nesta afirmação que o julgamento de o quão bom (ou ruim) é um dado de teste depende das características dos dados de teste executados anteriormente. Se o novo dado de teste exercitar algum elemento requerido já exercitado anteriormente, a qualidade deste dado de teste tende a ser menor, assim como o seu impacto no aumento da confiabilidade do software. Em outras palavras, o fato de exercitar um elemento requerido já exercitado diminui o potencial do dado de teste de gerar “nova informação” sobre o software em teste, em especial, informação sobre “novos defeitos” no software.

Ao se deslocar o foco da análise de dados de teste para *conjuntos* de dados de teste, chega-se à proposição de que os dados de teste do conjunto devem ser mutuamente complementares (e minimamente sobrepostos) levando a conjuntos de dados de teste representativos em relação ao domínio de entrada do software.

A lógica da conjectura acima é a seguinte: conforme já destacado, a propriedade de um dado de teste de exercitar algum elemento requerido ainda não exercitado contribui para o aumento de confiabilidade do software. Este ponto de vista “micro” se aplica a cada dado de teste. De um ponto de vista “macro”, para que cada dado de teste tenha a chance de contribuir para o aumento de confiabilidade do software e de agregar “nova informação” sobre o software, as diferenças entre os dados de teste devem ser promovidas. Em outras palavras, a diversidade de um conjunto de teste é, sob este ponto de vista “macro”, a

²⁶ Só é possível atingir 100% de cobertura caso todos os elementos requeridos sejam factíveis. Caso contrário não é possível atingir este valor. No caso de $N > P$ a cobertura máxima pode ser atingida mesmo com algumas partições tendo mais de um dado de teste.

propriedade desejável dos conjuntos de dados de teste, para que eles, ao serem executados, estabeleçam maiores níveis de confiabilidade do software.

Técnicas de Teste Baseadas em Particionamento

Esta seção apresenta conceitos sobre técnicas baseadas em particionamento e trabalhos que investigam a associação entre a eficácia do teste e a forma com que os defeitos se localizam no software. Mais especificamente, são descritos trabalhos que avaliam analiticamente a eficácia das técnicas baseadas em particionamento e o teste aleatório. É descrito também um trabalho que relaciona a incerteza sobre a localização de defeitos e a influência da diversidade dos dados de teste na eficácia das técnicas. Estes trabalhos permitem uma melhor compreensão sobre aspectos que influenciam a chance de revelar defeitos no software, ponto relevante quando se propõem novas maneiras para se selecionar dados de teste.

O termo “Teste Baseado em Particionamento” (*Partition Testing*) é usado para se referir a um amplo conjunto de técnicas de teste que dividem o domínio de entrada do programa em subdomínios (Hamlet e Taylor, 1990), (Weyuker e Jeng, 1991) (Chen e Yu, 1994). O testador seleciona um ou mais dados de teste de cada um desses subdomínios. Na literatura de teste o termo “partição” é utilizado para se referir aos subdomínios sem o rigor matemático do termo; isto é, os subdomínios (partições) podem ter regiões sobrepostas ou serem disjuntos. Essas técnicas também são chamadas de “Baseadas em Subdomínio” – *Subdomain Based Testing*, ex: (Ntafos, 1998), (Ntafos, 2001).

A título de exemplo, algumas técnicas baseadas em particionamento são citadas a seguir, detalhes em (Weyuker e Jeng, 1991).

Os critérios estruturais Todos os Nós e Todos os Arcos dividem o domínio de entrada em subdomínios não disjuntos. Cada subdomínio consiste de todos os dados de teste que causam a execução do o nó – ou arco. Um dado de teste que provoca a execução de vários nós e arcos encontra-se na interseção dos subdomínios associados a esses nós e arcos, fazendo parte, portanto de todos esses subdomínios. O critério Todos os Caminhos divide o

domínio de entrada em subdomínios disjuntos, pois cada dado de entrada provoca a execução de um caminho específico; cada subdomínio é formado por todos os dados de entrada que provocam a execução do caminho.

No caso de critérios baseados em análise de fluxo de dados o domínio de entrada é dividido de forma que existe um subdomínio correspondente a cada Associação Definição-Uso requerida pelo critério; um dado de entrada pode ser membro de vários subdomínios.

No critério Análise de Mutantes, os subdomínios consistem de dados de entrada que fazem com que um mutante seja distinguido do programa em teste. Algumas técnicas funcionais também fazem particionamento do domínio; por exemplo, o critério Particionamento em Classes de Equivalência define subdomínios que agrupam dados de teste tratados do mesmo modo segundo a especificação do software.

Até o Teste Exaustivo, um caso extremo, pode ser visto como uma técnica baseada em particionamento. O teste exaustivo requer que cada elemento do domínio de entrada seja testado; isto é, o domínio de entrada é dividido em subdomínios, cada um consistindo de um único dado de entrada – um ponto do domínio de entrada do programa.

Segundo Weyuker e Jeng (1991) o objetivo principal em se realizar o particionamento do domínio de entrada é de forçar o testador a selecionar dados de teste de maneira tal que o conjunto desses dados seja representativo de todo o domínio de entrada. Hamlet e Taylor destacam que o objetivo do particionamento é fazer com que cada subdomínio seja restrito o suficiente para que cada aspecto do programa, da especificação e do desenvolvimento seja separado em um único subdomínio (Hamlet e Taylor, 1990). Gutjahr pondera que, ao particionar o domínio de entrada, essas técnicas forçam o testador a diversificar os dados de teste a serem utilizados (Gutjahr, 1999).

Um ponto central em técnicas baseadas em particionamento é a capacidade da técnica de dividir o domínio de entrada de maneira tal que cada subdomínio tenha a propriedade que, para todos os elementos do subdomínio, ou o programa produza uma saída correta, ou produza uma saída incorreta. Subdomínios com esta propriedade são chamados de *reveladores* (Weyuker e Jeng, 1991) ou *homogêneos* (Hamlet e Taylor, 1990). Uma técnica

que sempre gerasse subdomínios reveladores seria perfeita, bastaria escolher um dado de teste de cada subdomínio para garantir que todos os defeitos fossem revelados. Na prática, no entanto, é extremamente difícil que os subdomínios sejam reveladores (Weyuker e Jeng, 1991). Hamlet e Taylor (1990) sugerem que na prática, ao se utilizarem técnicas que levam a partições não perfeitas (não reveladores), deve-se selecionar amostras de cada subdomínio em quantidade suficiente para aumentar a chance de revelar defeitos.

Eficácia do Teste Baseado em Particionamento em um Contexto Determinista

Diversos estudos têm sido conduzidos com o objetivo de ganhar entendimento sobre o que pode tornar técnicas de particionamento mais eficazes, assim como sobre qual o ganho em termos de eficácia do teste quando essas técnicas são comparadas com o teste aleatório.

Weyuker e Jeng abordam (1991) analiticamente a capacidade de revelar defeitos dessas técnicas, comparadas ao teste aleatório e investigam as circunstâncias as tornam mais eficazes ou menos.

Algumas definições são necessárias, detalhes em Weyuker e Jeng (1991). Assume-se que um programa P possua domínio de entrada D de tamanho d (número de possíveis dados de entrada) e com f pontos (dados de entrada) *causadores de falha*; todos os outros dados de entrada são *dados de entrada corretos* (não causam falhas quando são executados). Assume-se que n dados de teste sejam selecionados aleatoriamente com distribuição uniforme. $\Theta = f/d$ denota a taxa de falhas, isto é, a probabilidade de que um dado causador de falha seja selecionado como dado de teste. A probabilidade Pr de selecionar ao menos um dado de entrada causador de falhas quando n dados de teste são selecionados aleatoriamente é definida por:

$$Pr = 1 - (1 - \Theta)^n$$

Esta equação representa, portanto a chance de se provocar, durante o teste, a falha em um software que possui a taxa de falhas Θ por meio da seleção aleatória de dados de teste do domínio de entrada do software.

Quando um teste baseado em particionamento é realizado, o domínio de entrada do programa (**D**) é dividido em k subdomínios D_1, D_2, \dots, D_k , de tamanhos d_1, d_2, \dots, d_k , e que possuem números de pontos causadores de falhas f_1, f_2, \dots, f_k . A taxa de falhas de cada subdomínio i é $\theta_i = f_i/d_i$, resultando nas taxas falha $\theta_1, \theta_2, \dots, \theta_k$ associadas aos subdomínios $1, 2, \dots, k$.

A probabilidade **Pp** de selecionar ao menos um dado de entrada causador de falhas quando n_i dados de teste selecionados aleatoriamente de cada D_i é definida por:

$$Pp = 1 - \prod_{i=1}^k (1 - \theta_i)^{n_i}$$

Comumente é selecionado um dado de teste para cada subdomínio, ou seja, $n_1 = n_2 \dots = n_k = 1$. Deve-se notar que $d = \sum_{i=1}^k d_i = |D|$, ou seja, é assumido que os subdomínios não se sobrepõem; e $f = \sum_{i=1}^k f_i$, isto é, o total de pontos causadores de falhas é a soma desses pontos em cada um dos subdomínios.

Informalmente, **Pr** e **Pp** avaliam, respectivamente, a habilidade do teste aleatório e do teste baseado em particionamento em revelar defeitos no programa.

Análises e observações sobre circunstâncias que favorecem o teste aleatório e o teste baseado em particionamento são desenvolvidas. Situações destacadas em (Weyuker e Jeng, 1991) mostram que o teste baseado em particionamento pode ser melhor, pior, ou equivalente ao teste aleatório (em termos das probabilidades **Pr** e **Pp**). Embora os exemplos apresentados em sejam numéricos, apresentam-se a seguir situações, relacionadas aos exemplos numéricos, mas que apresentam qualitativamente as diferentes circunstâncias para a aplicação das técnicas.

Situação 1: Assumindo que um domínio **D** é dividido em k subdomínios, **Pp** é maximizado ($Pp = 1$) se um subdomínio contém apenas dados de entrada que produzem resultados incorretos. Portanto, a condição que maximiza a chance de revelar defeitos no teste baseado em particionamento é que pelo o menos um subdomínio contenha apenas pontos causadores de falha; neste caso $Pp \gg Pr$. Naturalmente, trata-se de uma situação irreal; se o testador tivesse o conhecimento necessário para agrupar os pontos de falha em um único subdomínio, não seria necessário realizar o teste, apenas remover os defeitos

causadores da falha, visto que ela já estaria identificada. Hamlet e Taylor, (1990) citam um cartunista (Walt Kelly) e satirizam esta situação: “para ter certeza de acertar o alvo, atire primeiro e então desenhe o alvo onde a seta estiver”. Porém, a Situação 1 mostra que quando uma técnica de teste particiona o domínio de entrada criando alguns subdomínios com alta chance de falha (valor alto para $\Theta = f/d$) e outros subdomínios com baixa chance de falha, a eficácia de técnicas baseadas em particionamento tende a ser expressivamente maior que a eficácia do teste aleatório ($Pp \gg Pr$). Nesta situação os subdomínios associados aos defeitos no software são próximos de serem *reveladores*, isto tende a acontecer quando a técnica permite agrupar em subdomínios conjuntos de dados que *realmente* são tratados uniformemente pelo software.

Situação 2: os subdomínios têm o mesmo tamanho (d) e o mesmo número de pontos causadores de falhas (f), portanto os subdomínios possuem a mesma taxa de falhas (Θ). Neste caso os valores Pr e Pp serão idênticos ($Pr = Pp$); o particionamento é ineficaz para aumentar a chance de revelar defeitos, pois ele não agrupou entradas causadoras de falhas – situação referida em (Gutjahr, 1999). como subdomínios “anti-reveladores”. Uma situação próxima da anterior – subdomínios de mesmo tamanho – mas com todos os pontos causadores de falha em um único subdomínio, faz com que $Pp > Pr$.

Situação 3: um subdomínio D_1 é expressivamente maior do que todos os outros ($D_1 \gg D_2, D_3, \dots, D_k$), além disso D_1 contém todos os pontos causadores de falhas (nenhum outro subdomínio tem pontos causadores de falhas). Esta situação ilustra o pior desempenho do teste baseado em particionamento. Intuitivamente isto ocorre porque essas técnicas forçam o testador a alocar dados de teste para todos os subdomínios menores e sem pontos causadores de falhas. O teste aleatório tem chance de alocar vários dados de teste no subdomínio D_1 , visto que ele é maior e, portanto tem mais chance de provocar falhas; ou seja, $Pr > Pp$. Realizar o particionamento e requerer o teste de vários subdomínios sem pontos causadores de falha significa, nesta situação, “desperdiçar” recursos de teste.

Embora corretas conceitualmente e úteis para avaliar teoricamente as técnicas de teste, as situações anteriores, conforme reconhecem os autores, são de pouca utilidade prática. O motivo é que o testador teria de saber as taxas de falhas associadas aos

subdomínios para identificar as situações descritas anteriormente. Em geral estas informações não são conhecidas, e menos ainda conhecidas com a certeza requerida pelo modelo analítico desenvolvido.

Eficácia do Teste Baseado em Particionamento: a Influência da Incerteza

Gutjahr (1999) em uma análise comparativa entre técnicas baseadas em particionamento e o teste aleatório mostra que, ao se considerar a *incerteza* sobre a taxa de falhas de cada subdomínio, o teste baseado em particionamento tende a ser amplamente mais eficaz do que o teste aleatório na maior parte dos casos. É utilizada a análise feita em Weyuker e Jeng (1991) já descrita; no entanto o modelo determinístico neste trabalho é estendido para um modelo probabilístico no qual as taxas de falhas de cada subdomínio são tratadas de forma probabilística e modeladas como variáveis aleatórias.

O argumento de Gutjahr (1999) é que as taxas de falhas associadas aos subdomínios nunca são conhecidas com certeza; isto é ilustrado nas afirmativas de um testador hipotético: “o programa em teste tem com certeza taxa de falhas de 2 %”, ou “o programa em teste é uma instância de uma classe maior de programas similares, para a qual dados empíricos sobre taxa de falhas e parâmetros estatísticos estão disponíveis”. Segundo o autor a segunda afirmativa é certamente mais factível do que a primeira; fato que motiva a extensão da análise em Weyuker e Jeng (1991) considerando um modelo probabilístico das taxas de falhas.

Os resultados de Gutjahr mostram que esta abordagem probabilística muda o cenário a favor das técnicas baseadas em particionamento. Uma analogia é que em ciências de gerenciamento a consideração da incerteza é um aspecto de grande importância, e que leva a diferentes decisões ótimas em comparação com as decisões em situações absolutamente certas. Por exemplo, decisões sobre a alocação de recursos financeiros (portfólio) em cenários de indecisão, sempre tornam a diversificação de investimentos a melhor estratégia. Na questão da seleção de dados de teste a incerteza sobre a taxa de falhas – e a localização dos defeitos – faz com que a estratégia de diversificação dos dados de teste esteja relacionada à eficácia do teste.

A extensão feita por Gutjahr em relação a Weyuker e Jeng consiste essencialmente em considerar os *valores esperados* das taxas de falhas para subdomínios ao invés de considerar as taxas de falhas. Com isto as fórmulas para a probabilidade de detecção de defeitos também são expressas em termos de expectativas com respeito às distribuições de probabilidade das variáveis aleatórias que caracterizam as taxas de falhas consideradas. Portanto, as equações de Weyuker e Jeng são redefinidas para considerar que os valores P_p e P_r são esperados para probabilidade de detecção de falhas.

São mostrados exemplos hipotéticos de alocação de dados de teste no teste baseado em particionamento considerando duas situações²⁷: na situação 1 há certeza sobre as taxas de falhas do programa em teste, enquanto que na situação 2 as taxas de falhas não são conhecidas, mas os seus valores esperados são conhecidos. Na situação 1, portanto, o testador recebe informação, com nível máximo de certeza, sobre as taxas de falhas de subdomínios e pode alocar dados de teste para estes subdomínios com base nesta informação. É mostrado que a decisão ótima neste caso consiste em alocar todos dos dados de teste ao subdomínio com maior taxa de falhas. Nesta situação, esta alocação de dados de teste maximiza a chance de encontrar defeitos (valor de P_r). Na situação 2, na qual as taxas de falhas não são certas, mas sim estimadas, a alocação ótima de dados de teste (que maximiza P_r) muda. Nesta segunda situação a alocação de dados que leva a um aumento da chance de encontrar defeitos é a que exercita todos os subdomínios; A conclusão desta análise é que a incerteza sobre as reais taxas de falhas favorece a estratégia de diversificação.

Gutjahr analisa em especial a situação em que o testador não tem informações sobre subdomínios mais sujeitos a defeitos e, portanto estima a mesma taxa de falhas para todos os subdomínios; deve-se notar que isto não implica que as taxas de falhas reais sejam iguais. Nesta comparação assume-se que o número de dados de entrada causadores de falha em cada subdomínio é representado por variáveis aleatórias independentes. É provado que o teste baseado em particionamento é melhor ou equivalente ao teste aleatório ($\overline{P_r} \leq \overline{P_p}$). O limite inferior para P_r é definido pela equação a seguir, onde k é o número de subdomínios,

²⁷ Notar que as situações descritas nesta Seção não são as mesmas da Seção anterior.

$\bar{\theta}$ é o valor esperado para taxa de falhas, e \bar{P}_r e \bar{P}_p são respectivamente os valores esperados para probabilidade de revelar defeitos com o teste aleatório e com o teste baseado em particionamento.

$$\bar{P}_r \geq \frac{\bar{\theta}}{1-(1-\bar{\theta})^k} \bar{P}_p$$

Isto mostra que, para valores baixos para a taxa de falhas esperada, o limite inferior para P_r aproxima-se de P_p / k , ou seja, o teste aleatório pode ser expressivamente menos eficaz do que o teste baseado em particionamento. As circunstâncias em que este limite inferior é atingido são:

- a) Existem muitos subdomínios pequenos e poucos subdomínios grandes; e
- b) Os subdomínios são próximos de serem *reveladores* (no sentido definido anteriormente).

Os autores argumentam ainda que estas premissas são razoáveis de serem satisfeitas na prática, o que leva a eficácia do teste aleatório tender mais para o limite inferior do que para o limite superior (equivalente ao teste baseado em particionamento) em situações reais.

Deve-se notar que as circunstâncias em que $P_p > P_r$ identificadas em Gutjahr (1999) são similares às identificadas a Weyuker e Jeng (1991), no entanto, resultados identificados em Gutjahr (1999) são únicos e importantes para a compreensão sobre técnicas de teste.

Um ponto importante é que as situações a) e b) anteriores não seriam tão raras. Na verdade, segundo o autor seria mais comum que subdomínios tenham tamanhos desiguais (situação a) do que tenham o mesmo tamanho. Além disso, é mais provável que taxas de falhas estejam mais próximas do caso “revelador” do que do “anti-revelador” – a distribuição de taxas de falha tenderia a uma forma “U”, com mais subdomínios com taxas ou mais altas ou mais baixas, e menos subdomínios com taxas de falhas médias.

Sumario: Técnicas Baseadas em Particionamento e Diversidade

O uso de técnicas baseadas em particionamento pode ser visto como um meio de forçar o testador a selecionar dados de teste que sejam representativos do domínio de entrada do programa sob a ótica da técnica utilizada. Embora a eficácia dessas técnicas dependa de aspectos, tais como o número e os tamanhos das partições e a forma com que os defeitos estão alocados às partições, os resultados das análises indicam que as técnicas baseadas em particionamento tendem a ser melhores que o teste aleatório. Importante ressaltar que as boas técnicas (em relação à eficácia) tendem a agrupar entradas causadoras de falha enquanto as técnicas ruins tendem a gerar partições “anti-reveladoras”, com os defeitos distribuídos pelas partições.

Um ponto importante e diretamente relacionado às técnicas propostas nesta tese é a interpretação de que técnicas baseadas em particionamento induzem a uma diversificação dos dados de teste. Esta diversificação torna-se vantajosa para a eficácia do teste quando as taxas de falha dos subdomínios – ou analogamente a localização dos defeitos – são desconhecidos. A existência de uma relação entre diversidade e eficácia do teste é uma hipótese razoável na prática, dado que na maioria das situações, realmente não há com o testador determinar a priori as taxas de falhas associadas aos subdomínios. Conforme destaca Gutjahr presumir taxas de falhas iguais para os subdomínios é natural, até mesmo porque a prática comum de se selecionar o mesmo número de dados de teste para cada subdomínio denota, implicitamente, que não há informação de que um subdomínio tem mais defeitos do que outros. Na presença de informação deste tipo o mais natural seria alocar mais dados de teste para subdomínios com mais defeitos.

Esta relação entre diversidade e eficácia pode ser vista como um estímulo à idéia de desenvolver técnicas de teste que tenham a diversidade como o alvo do teste, ao invés da diversidade surgir apenas como um “efeito colateral” das técnicas baseadas em particionamento.

Diversidade no Sistema Imune

Conforme já descrito no Capítulo 4, o *Sistema Imune* (ou *Sistema Imunológico*) é responsável por proteger seres vivos de entidades perigosas internas e externas (Patógenos) representados por microorganismos causadores de doenças tais como Vírus, Bactérias, Fungos ou Parasitas (De Castro, 2001), (De Castro e Timmis, 2002).

Os sistemas imunes (inato e adaptativo) possuem mecanismos de identificação capazes de distinguir células próprias saudáveis de patógenos (ou *antígenos*). Os linfócitos produzem anticorpos, que são capazes de reconhecer e eliminar os patógenos, sendo que cada anticorpo é específico para reconhecer um tipo de patógeno,

Os recursos para esta atuação do sistema imune são limitados: os tipos de anticorpos do repertório são finitos, no entanto, este sistema precisa reconhecer um número potencialmente infinito de patógenos e células cancerosas.

Memória e Capacidade de Generalização

Estratégias do sistema imune compreendem as capacidades de memorizar, de apreender e de generalizar no reconhecimento desses patógenos. Isto permite o aprimoramento contínuo da resposta imunológica ao longo da vida do indivíduo. Quando um indivíduo é exposto a um patógeno pela primeira vez, ocorre uma reação por meio da geração de anticorpos específicos para o patógeno. Esta primeira reação (*resposta primária*) ocorre, digamos, em um tempo de reação $tr1$ e com uma concentração $cr1$ de anticorpos. Em uma segunda exposição ao mesmo patógeno, ocorre uma *resposta secundária* com tempo de reação $tr2$ e com uma concentração $cr2$ de anticorpos. Esta resposta secundária tende a ser mais rápida ($tr2 < tr1$) e mais efetiva do que a primeira ($cr2 > cr1$). Em outras palavras, o sistema imune “aprende” a responder ao patógeno. Caso o patógeno da segunda exposição seja *semelhante* ao da primeira exposição – mas não idêntico a ele – ainda assim a *resposta secundária* tende a ser aprimorada, o que revela a capacidade de generalização deste sistema (De Castro, 2001), (De Castro e Timmis, 2002).

Mecanismos de Promoção de Diversidade

Um aspecto fundamental do sistema imune – e particularmente interessante no contexto deste trabalho – é a existência de mecanismos para criar diversidade, por meio dos quais este sistema aumenta a confiabilidade da sua atuação (Oprea e Forrest, 1999). Conforme já destacado, cada linfócito produz anticorpos de atuação específica, definida pelos seus receptores. Os linfócitos – desenvolvidos na Medula Óssea ou no Timo – têm os seus receptores determinados por um mecanismo de rearranjo gênico, que atua para gerar centenas de diferentes variantes dos genes codificados nas moléculas receptoras. Segundo Berek e Milstein (1988) humanos possuem cerca 10^5 genes no genoma, enquanto o sistema imune pode produzir cerca de 10^{11} diferentes moléculas de anticorpos. Só assim podem ser produzidos milhões de linfócitos em um organismo, que possuem milhões de especificidades distintas.

Outro mecanismo que induz a uma alta diversidade de anticorpos são as hipermutações somáticas que ocorrem quando há estímulo de antígenos (De Castro, 2001). Quando um antígeno é reconhecido por um anticorpo há a proliferação (ou clonagem) deste anticorpo; neste processo, células descendentes são geradas, algumas contendo alterações (inserções e deleções de genes) nos receptores resultantes de mutações aleatórias. Esta variedade de receptores com pequenas alterações permite refinar a resposta imunológica ao antígeno, além de aprimorar a “capacidade de generalização” descrita anteriormente.

Esses dois mecanismos, rearranjos gênicos e hipermutações, fazem com que o sistema imune seja capaz de produzir uma quantidade praticamente infinita de receptores celulares a partir de um genoma finito. Essa estratégia pode ser analisada sob a perspectiva do conceito de *Espaço de Formas* (Perelson e Oster, 1979) que modela de maneira abstrata as interações moleculares entre as células imunes e os antígenos.

Espaço de Formas, Afinidade e Reconhecimento Imune

O *Espaço de Formas* define um espaço multidimensional no qual a localização de anticorpos e de antígenos é determinada por suas características físico-químicas (ou

propriedades) tais como: carga eletrostática, estrutura geométrica, ligações de hidrogênio, dentre outras. Neste modelo, cada dimensão do espaço refere-se a uma característica (ou *propriedade*) relevante do anticorpo, para a qual valores do tipo real podem ser atribuídos²⁸ (Perelson e Oster, 1979), (Hightower et al, 1995), (De Castro, 2001). Cada anticorpo (e também cada antígeno) é, portanto, representado como um *ponto* no espaço de formas cuja posição depende dos valores atribuídos para cada propriedade do anticorpo (ou antígeno).

A *distância euclidiana* entre um anticorpo e um antígeno é uma medida da *afinidade* entre eles; esta afinidade é determinada pela força de interação entre as moléculas e caracteriza a chance de que o anticorpo reconheça o antígeno. Uma *esfera de ativação* existe ao redor de anticorpos e define os limites até os quais é possível o reconhecimento de antígenos pelos anticorpos. Ou seja, *esfera de ativação* define um escopo de atuação dentro do qual qualquer antígeno poderá ser reconhecido pelo anticorpo. A Figura 38 original de Perelson e Oster (1979) ilustra o espaço de formas S e uma região V preenchida com anticorpos. Cada ponto representa um anticorpo e cada x representa um antígeno. É destacado um anticorpo e representada a sua esfera de ativação de raio ϵ . Este anticorpo é capaz de se ligar a qualquer antígeno que esteja dentro desta esfera, dois antígenos nesta figura.

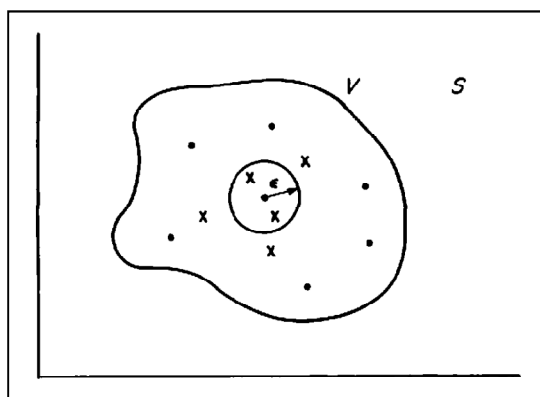


Figura 38. Ilustração do modelo espaço de formas, Perelson e Oster (1979)

O modelo de Perelson e Oster (1979) considera fatores como: o tamanho e número de dimensões *espaço de formas*, o número de anticorpos do repertório, o volume das *esferas*

²⁸ Espaços de formas podem também ser definidos usando valores binários e medidas de distância Hamming.

de ativação, dentre outros, e estabelece condições que o sistema imune deve satisfazer para que os antígenos sejam detectados de forma confiável. Uma conclusão interessante destacada é que “a probabilidade de reconhecer uma molécula externa aumenta monotonicamente com o tamanho do repertório, mas que tamanhos de repertório grandes são necessários para que se alcance uma boa probabilidade de reconhecimento bem sucedido”.

A grande eficácia do sistema imune para o combate às entidades perigosas leva à hipótese, levantada por pesquisadores em imunologia, de que este sistema aloca estrategicamente anticorpos no espaço de formas, maximizando a cobertura deste espaço. Alguns trabalhos investigam esta hipótese por meio de simulações computacionais, elaboradas com base em modelos do sistema imune.

A Estratégia de Cobertura do Sistema Imune

Hightower et al. (1995) utilizam um modelo abstrato do sistema imune (SI) e aplicam um algoritmo genético (AG) para estudar a natureza da organização genética resultante da evolução dos genes do sistema imune. Mais especificamente, foi avaliada a evolução de moléculas de anticorpos que são responsáveis pelo reconhecimento dos antígenos.

Cada indivíduo do AG representa a especificação genética para as bibliotecas de anticorpos de um sistema imune. Para simular o sistema imune humano, o genótipo do indivíduo é dividido em quatro bibliotecas. Cada biblioteca possui oito segmentos de anticorpos. Cada um desses segmentos, por sua vez, é representado como um string de 16 bits²⁹. O processo de expressão (ou formação) dos anticorpos é feito pela seleção aleatória de um segmento de cada uma das quatro bibliotecas; os quatro segmentos selecionados são então concatenados para formar o anticorpo. Como cada segmento é um string de 16 bits, um anticorpo expresso é representado como um string de 64 bits.

²⁹ Cada segmento de anticorpo é representado como um string de 16 bits, portanto cada biblioteca tem 128 (16x8) bits. Considerando todas as 4 bibliotecas, tem-se que um indivíduo é representado por 512 (128x4) bits.

A partir das quatro bibliotecas de segmentos de anticorpos, vários diferentes anticorpos podem ser formados (ou expressos), dependendo de quais segmentos forem aleatoriamente selecionados das bibliotecas. O conjunto dos anticorpos que podem ser construídos a partir das bibliotecas de segmentos é o *repertório potencial de anticorpos*. O conjunto de anticorpos que são expressos (efetivamente construídos) é chamado *repertório de anticorpos expressos*. O número desses anticorpos expressos (referido como N) é um parâmetro do modelo usado para o SI. Outro parâmetro do modelo é o número de antígenos aos quais o indivíduo será exposto, referido como K . Os autores utilizaram diferentes valores para N e K nos experimentos.

A *afinidade* entre um anticorpo e um antígeno – que caracteriza a chance de que o anticorpo reconheça o antígeno – é computada pelo número de “encaixes” bit a bit (valores complementares para os bits) entre o antígeno e o anticorpo.

O *fitness* (ajuste) de um indivíduo do AG caracteriza a sua habilidade de reconhecer os antígenos a ele expostos. O *fitness* do indivíduo é calculado por meio da exposição do indivíduo a um conjunto de K antígenos e da avaliação de o quão bem ele reconhece cada antígeno do conjunto. Os anticorpos expressos do indivíduo são responsáveis pela tarefa de reconhecimento. Cada antígeno (um do total K) recebe um *score* que é o máximo de todas as afinidades computadas entre este antígeno e todos os N anticorpos selecionados para a expressão. O *fitness* do indivíduo é então computado como a média dos *scores* para os diferentes antígenos.

Notar que, ao considerar o máximo das afinidades computadas para um antígeno, o modelo faz com que indivíduos com valores maiores de *fitness* sejam aqueles que conseguem que algum anticorpo expresso tenha uma boa afinidade com cada antígeno exposto para reconhecimento. Observar também que a seleção dos N anticorpos a serem expressos, assim como dos K antígenos a serem reconhecidos, é feita usando mecanismos que envolvem aleatoriedade. Embora não esteja explícito, o artigo sugere que os K antígenos são selecionados no início da simulação e permanecem os mesmos ao longo das iterações do AG.

O AG foi aplicado para evoluir os indivíduos alocando maior chance de seleção a indivíduos com maior *fitness* e aplicando operadores genéticos de recombinação e mutação.

Os resultados das simulações sugerem que as bibliotecas de anticorpos dos indivíduos tornam-se progressivamente mais dissimilares, mesmo considerando que esta dissimilaridade não é requerida explicitamente na função de ajuste. Os autores destacam que este “efeito emergente” representa uma alocação balanceada de anticorpos para a tarefa de reconhecer os antígenos. A explicação é que, quando a distância entre dois anticorpos é pequena, estes anticorpos irão reconhecer os mesmos antígenos; ou seja, anticorpos similares terão uma cobertura sobreposta do espaço de antígenos. Esta cobertura sobreposta é redundante e potencialmente reduz a capacidade geral do sistema imune de reconhecerem os antígenos.

Dada a limitação de recursos do sistema imune, é desejável que coberturas sobrepostas sejam reduzidas pela alocação de anticorpos distantes uns dos outros. Nesta alocação, quando os anticorpos estão maximamente distantes uns dos outros, a cobertura máxima do espaço de antígenos é atingida. Esta hipótese foi investigada pela análise da “Separação Hamming” que mede a distância média entre pares de anticorpos. O valor desta medida aumenta ao longo da evolução dos anticorpos realizada pelo AG e tem um relacionamento próximo de linear com os valores de *fitness* obtidos. A conclusão é que a tarefa de reconhecer antígenos pode ser considerada como um problema de cobertura de espaços, e que foram obtidas evidências de que o Sistema Imune evolui de forma emergente a estratégia de aprimorar a diversidade dos anticorpos do repertório, visando aumentar a capacidade de reconhecer antígenos (Hightower et al, 1995).

Oprea e Forrest (1998, 1999) também investigam o que eles chamam de “capacidade antecipatória” do sistema imune, ou seja, a capacidade deste sistema em detectar patógenos para os quais o sistema não foi treinado para reconhecer. Assim como em Hightower et al um algoritmo evolutivo é utilizado para explorar as estratégias que as bibliotecas de anticorpos evoluem frente à tarefa de reconhecer conjuntos de patógenos de diferentes tamanhos.

Anticorpos – assim como os patógenos – são representados como strings de bits e evoluem submetidos a operações de recombinação de n pontos e mutação simples. A simulação realizada captura a ideia de que, para cada patógeno no ambiente, pelo menos um anticorpo da biblioteca de anticorpos do indivíduo deve reconhecer o patógeno. Portanto, o escore do indivíduo com respeito a um patógeno p é proporcional à distância Hamming entre p e o anticorpo mais próximo de p (com menor distância *Hamming* de p). O *fitness* do indivíduo é calculado como a média dos escores com respeito a todos os patógenos. A seleção de indivíduos é baseada em um ranque feito com base nos valores de *fitness* dos indivíduos.

Os resultados de Oprea e Forrest confirmam a análise de Hightower et al. e apontam também a existência de uma relação entre o tamanho do conjunto de patógenos aos quais o indivíduo é exposto e a estratégia de cobertura do sistema imune. Quando o conjunto de patógenos é relativamente pequeno, este conjunto determina diretamente a estrutura das bibliotecas de anticorpos. Neste caso os anticorpos tendem a reconhecer os patógenos de forma perfeita, fazendo com que a biblioteca de receptores reflita diretamente a estrutura do conjunto de patógenos. Ao contrário, quando este conjunto é muito maior que o número de anticorpos disponíveis, a biblioteca de anticorpos evolui para garantir a máxima cobertura do espaço de possíveis patógenos. Neste caso, a estrutura dos anticorpos torna-se independente do conjunto de patógenos expostos, para os quais ela evoluiu visando reconhecê-los. Portanto, quando se aumenta o número de patógenos a serem reconhecidos, o sistema imune muda a sua característica de ser completamente determinado por eles, para a estratégia de cobrir tão completamente quanto possível o espaço de possíveis patógenos (Oprea e Forrest, 1998).

Segundo os autores, esta cobertura completa se dá pelo aumento da distância Hamming média entre pares de anticorpos da biblioteca, que tende a ser maior do que esta medida em bibliotecas de anticorpos geradas aleatoriamente. Tal estratégia de cobertura é responsável pela “capacidade antecipatória” do sistema imune.

Resumo: um Paralelo Entre o Reconhecimento Imune e a Revelação de Defeitos

Esta seção apresentou mecanismos de preservação da vida relacionados à diversidade. Foram descritas características do sistema imune e foi destacado o papel da diversidade de anticorpos no sucesso ao combate de agentes perigosos. Em especial, buscou-se mostrar trabalhos que analisam a estratégia utilizada pelo sistema imune para cobrir o espaço de possíveis patógenos.

O reconhecimento imune ocorre quando existe algum anticorpo no espaço de formas próximo ao ponto onde está o antígeno (patógeno) a ser combatido, ou seja, existe um anticorpo com epitopo específico para reconhecer o antígeno. A distância limite para que aconteça o reconhecimento (limiar de afinidade) é definido pelo raio da esfera de atuação do anticorpo. Além deste valor, o reconhecimento imune não ocorre.

A revelação de um defeito ocorre quando existe algum dado de teste localizado em um ponto do domínio de entrada onde exista uma região de falha associada a um defeito no software. Para o dado de teste não há uma “esfera de atuação”, isto é, a execução do software com um dado de teste (que define um ponto do domínio de entrada) avalia o comportamento do software (falha ou funcionamento correto) apenas neste ponto específico. No entanto, conforme destacado no Capítulo 4, a continuidade de regiões de falha faz com que haja uma “esfera de atuação” do teste, no sentido de que o dado de teste pode estar em qualquer ponto da região de falha para que o defeito seja revelado.

Importante notar que o sistema imune apresenta um conjunto de mecanismos elaborados e que a estratégia de aprimorar a diversidade é apenas um desses mecanismos. Esta analogia entre a atuação do sistema imune e o teste de software foi desenvolvida devido ao fato de as estratégias deste sistema terem inspirado o desenvolvimento das ideias da geração de dados orientada à diversidade, descrita no Capítulo 4.

A Tabela 13 sintetiza a analogia entre elementos relativos ao sistema imune e elementos relativos ao teste de software.

Tabela 13. Analogia entre o reconhecimento imune e a revelação de defeitos

Conceitos e mecanismos do Sistema Imune	Conceitos e mecanismos do Teste de Software
Os mecanismos do sistema imune visam a identificar e combater entidades perigosas, permitindo assim a manutenção da vida do indivíduo.	A atividade de teste de software visa a revelar a presença de defeitos no software, permitindo a remoção dos defeitos.
Anticorpo: agente fundamental da ação imune.	Dado de teste: elemento fundamental para execução do teste.
Espaço de formas.	Domínio de entrada do programa.
Anticorpos são pontos no espaço de formas.	Dados de teste são pontos no domínio de entrada.
Reconhecimento Imunológico.	Revelação do defeito.
Esfera de ativação: define a região do espaço de formas em que um anticorpo é capaz de reconhecer antígenos.	Região de falha: define a região do domínio de entrada em que um dado de teste é capaz de provocar a falha.
Cada dimensão do espaço de formas representa uma propriedade relevante para o reconhecimento de patógenos.	Cada dimensão do domínio de entrada do programa representa uma variável de entrada relevante para a revelação de defeitos.
Repertório Imunológico: conjunto de agentes responsáveis pela eficácia da resposta imune.	Conjunto de dados de teste: conjuntos de elementos responsáveis pela eficácia do teste.
Patógeno (Antígeno): elemento a ser reconhecido. Almeja-se que algum anticorpo do repertório seja capaz de reconhecer um patógeno invasor.	Defeito no software: elemento a ser identificado. Almeja-se que algum dado de teste do conjunto seja capaz de revelar o defeito.
Incerteza em relação a qual patógeno irá invadir o corpo: incerteza em relação à localização de patógenos no espaço de formas.	Incerteza em relação a quais defeitos estão presentes em um software: incerteza em relação à localização de regiões de falha no domínio de entrada.
Estratégia para aumentar a chance do reconhecimento imune: mecanismos promovem a diversidade de anticorpos do repertório imune.	Estratégia para aumentar a chance de revelar defeitos: mecanismos promovem a diversidade de dados de teste do conjunto de teste.

Apêndice D – Esboço da perspectiva de diversidade DOTG – *Execution Information*

Este Apêndice detalha o trabalho futuro de desenvolver e validar outras perspectivas para a diversidade. É fornecido um esboço da perspectiva que leva em conta informações sobre a execução do software durante o teste (*DOTG-EI, EI: Execution Information*),

D.1 Exemplo: a perspectiva *DOTG-EI, EI: Execution Information*

Uma possibilidade de estruturação de informações para a DOTG-EI utilizando o conceito de Associação Definição-Uso (ADU) é descrita a seguir:

Observações:

- ADU é uma associação definição-uso (p-uso ou c-uso) com respeito uma ou mais variáveis do programa;
- T1 e T2 são conjuntos de teste;
- t_i representa um dado de teste;
- $(t_i - ADU_i)$ significa que o dado de teste t_i do conjunto exercitou a ADU_i ;
- $(t_i - ADU_i, ADU_j)$ significa que o dado de teste t_i do conjunto exercitou a ADU_i e a ADU_j).

Nível 1: Associações de fluxo de dados (ADU) exercitadas.

Exemplo 1:

T1: $\{(t1 - ADU1), (t2 - ADU3), (t3 - ADU2)\}$

T2: $\{(t1 - ADU1), (t2 - ADU1), (t3 - ADU2)\}$

Explicação: T1 tem maior diversidade que T2, pois exercitou mais ADUs. Este nível é equivalente a exercitar associações definição-uso no critério de teste Todos os Usos. T1 tem maior cobertura (diversidade) que T2.

Observação: notar que a meta-heurística busca otimizar (neste caso maximizar) o valor objetivo. Neste caso, isto significa encontrar dados de teste que maximizem a cobertura de ADUs. A meta-heurística não “sabe de antemão” necessariamente quais são essas ADUs.

Nível 2: Diferentes combinações em que ADUs são exercitadas.

Pode-se dizer que uma execução do software gera um “evento de fluxo de dados” (EFD) representado pela sequência de ADUs exercitadas pelo dado de teste. A inserção de qualquer EFD diferente dos já existentes em conjunto de teste significa um aumento de diversidade do conjunto de teste.

Exemplo 2:

T1: {(t1 – ADU1), (t2 – ADU3), (t3 – ADU2), (t4 – ADU4), (t5 – ADU2)}

T2: {(t1 – ADU1), (t2 – ADU3), (t3 – ADU2, ADU3), (t4 – ADU4), (t5 – ADU2)}

Explicação: T1 e T2 alcançam a mesma cobertura do critério de teste Todos os Usos. Contudo, em T1 t3 e t5 são equivalentes. Em T2 t3 não é equivalente à t5, pois exercita também ADU3. Deste modo, a diversidade de T2 é maior do que a de T1.

Notar que a situação de um dado de teste exercitar mais de uma ADU é comum, e ocorre quando o dado de teste está em uma região do domínio de entrada associada à interseção de dois subdomínios, de duas ADUs distintas. O efeito deste nível 2 é tratar interseções entre ADUs como um “elemento requerido adicional” para distinguir a maior diversidade de T2 em relação à T1.

Considerando os dados de teste tx e ty: (tx – ADUn, ADUj) e (ty – ADUj, ADUn), apesar de tx e ty exercitarem as mesmas ADUs, a ordem de cobertura é diferente. Esta situação, provocada por ADUs dentro de laços, também seria tratada no Nível 2.

A explicação para este nível faz sentido, pois um “evento de fluxo de dados” diferente dos existentes tem chance de revelar algum defeito maior do que zero (por exercitar os fluxos de controle e de dados de uma forma ainda não exercitada). Isto não ocorre com dados de entrada duplicados.

Nível 3: Diferentes valores com que ADUs, ou combinações de ADUs, são exercitadas.

Exemplo 3:

$$\left. \begin{array}{l} T1: \{(t1 - ADU1), (t2 - ADU2), (t3 - ADU2)\} \quad T1: Div(t2,t3) = N \\ T2: \{(t1 - ADU1), (t2 - ADU2), (t3 - ADU2)\} \quad T2: Div(t2,t3) = M \end{array} \right\} \boxed{N > M}$$

Explicação: Em ambos os conjuntos de teste, os dados de teste t2 e t3 exercitam a mesma ADU (ADU2). Supondo que a ADU2 seja com respeito a uma variável x. Div(t2,t3) retorna a diversidade de valores para x tomados no ponto onde x é definida (ou equivalentemente onde x é usada) e considerando t2 e t3. Neste caso, a diversidade de valores de x em T1 (N) é maior do que a diversidade de valores de x em T2 (M). Isto significa que, embora tanto T1 quanto T2 tenham dois dados de teste equivalentes (t2 e t3), a diversidade de T1 é maior porque a diferença entre t2 e t3 em termos de estados de execução (valores de variáveis) é maior em T1.

Neste exemplo apenas dois dados de teste (t2 e t3) exercitam a mesma ADU, mas valores de diversidade podem ser calculados quando vários dados de teste exercitam a mesma ADU: Div(t1, t2, ..., tn). Se a ADU for com respeito a mais de uma variável (ex: variáveis x, y, z) os valores de todas as variáveis devem ser considerados para calcular Div. A função que calcula a diversidade pode ser baseada na distância euclidiana, assim como a função Div utilizada na técnica DOTG-ID (Capítulo 4).

A explicação para este nível faz sentido, pois é possível exercitar uma parte do código defeituosa e não revelar o defeito. O aumento da diversidade entre os valores t2 e t3 utilizados para exercitar a ADU2 aumenta a chance de revelar “regiões de erro” que possam existir.

A Figura abaixo (já utilizada no Capítulo 4) ilustra este fato. O defeito na condição do comando if só cria uma avaliação incorreta do predicado se a expressão $(2 * X - Y)$ assumir valores entre 10 e 18. Os dados de teste (X,Y) d1= (2,3) e d2= (20,3) exercitam todas as AFDs e não revelam o defeito. Neste caso, aumentar a diversidade dos dados de teste (pares x,y) que alcançam o comando defeituoso aumenta a chance de revelar o defeito (comparado a exercitar dois dados de teste idênticos).

```
int X,Y,Z;
do{
    scanf("Enter X,Y: %d %d", X,Y);
} while (X<=0||Y<=0);
If (2 * X - Y > 10)
    /*comando correto: If (2 * X - Y > 18)*/
    Z = X / 2 * Y;
else
    Z=X*Y;
print("%d",Z);
```

Figura 39. Exemplo de diversidade de estados internos de execução

Outra maneira de calcular a diversidade de valores de dados seria associar às ADUs histogramas que capturam os valores das variáveis em cada execução e permitem direcionar a busca pela diversidade de estados (valores). Esta abordagem é semelhante à adotada por Alshraideh et al (2010) para buscar dados escassos.

Os três níveis descritos acima podem ser utilizados formar um a função objetivo e uma meta-heurística pode ser utilizada para otimizar os valores de diversidade dos conjuntos de teste. Pode-se conjecturar sobre as características dos conjuntos DOTS associados à DOTG-IE:

- Quanto mais Associações Definição-Uso (ADU) exercitadas, maior a diversidade do conjunto;
- Quando mais Eventos de Fluxo de Dados (EFD) exercitados, maior a diversidade do conjunto;

- Considerando que mais de um dado de teste exercita uma mesma ADU ou um mesmo EFD, quanto maior a diversidade de valores das variáveis envolvidas na ADU (ou no EFD), maior a diversidade do conjunto;

Se na técnica DOTG-ID a diversidade é avaliada em relação à cobertura de dados de teste no domínio de entrada, na técnica DOTG-IE, a diversidade é avaliada em relação a como o código do software é exercitado, com ênfase na cobertura de elementos de fluxo de dados e de valores internos de variáveis (estados internos da execução). Portanto, se na técnica DOTG-ID almeja-se encontrar regiões de falha no domínio de entrada, na técnica DOTG-EI deseja-se encontrar regiões de erro, isto é, regiões associadas a estados internos de execução incorretos, que podem levar a uma falha (veja seção 4.9).

Outro ponto: na DOTG-ID todas as regiões do domínio de entrada são tratadas igualmente em termos de alocação de dados de teste. Já na DOTG-EI as regiões do domínio de entrada associadas a uma maior complexidade (em termos de fluxo de dados) vão ser mais exercitadas.

D2 Perspectiva DOTG-EI, EI: Execution Information – Outra direção

Outra direção a ser explorada para a técnica DOTG-IE é considerar cada Associação Definição-Uso como uma categoria e caracterizar a diversidade pela maneira com que os elementos (dados de teste) estão distribuídos pelas categorias (ADU). Nesta direção, Índices de diversidade podem ser aplicados para calcular a diversidade na alocação de um conjunto de dados de teste para um conjunto de ADUs.

Por exemplo, o Índice de Shannon (1948) pode ser usado:

$$H' = - \sum_{i=1}^R p_i \log p_i$$

Este índice, proposto originalmente para quantificar a entropia – ou incerteza – em um conteúdo de informação, é aplicado, por exemplo, para medir a biodiversidade de ecossistemas. Neste contexto H' é o valor da biodiversidade, R é o número de espécies

encontradas no ecossistema, e p_i é a proporção de indivíduos encontrados da espécie i em relação ao total de indivíduos.

H' aumenta com o número de espécies encontradas (R) considerando todos os indivíduos. Para um mesmo valor de R , H' aumenta na medida em que os valores p_i se tornam mais próximos para todo i . Isto é, quanto mais espécies encontradas mais diversidade. Considerando um determinado número de espécies encontradas a diversidade é maior quanto mais próximos forem os números de indivíduos de cada espécie (p_i).

A aplicação do Índice Shannon para geração de dados de teste orientada à diversidade faria com que as meta-heurísticas gerassem conjuntos DOTS com o seguinte direcionamento: (i) quanto maior o número de ADUs exercitadas, maior a diversidade; (ii) considerando um mesmo número de ADUs exercitada, quanto mais uniforme for a alocação de dados para as ADUs, maior a diversidade. Ou seja, a busca tenderia a prioritariamente aumentar a cobertura de ADUs e, na medida em que a cobertura aumentasse, haveria a tendência de alocar os dados de teste de maneira mais uniforme entre as ADUs.

Referências

Abreu, B.T.; Martins, E.; e Sousa, F.L., “Generalized extreme optimization: An attractive alternative for test data generation,” *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '07)*, Morgan Kaufmann Publishers Inc., USA, pp 1138, 2007.

Acree, A.T.; “On Mutation,” *PhD Thesis, Georgia Institute of Technology*, Atlanta, Georgia, 1980.

Afzal W.; Torkar, R.; e Feldt, R.; 2009. “A systematic review of search-based testing for non-functional system properties,” *Inf. Softw. Technol.* 51, 6 (Junho 2009), 957-976. DOI=10.1016/j.infsof.2008.12.005 <http://dx.doi.org/10.1016/j.infsof.2008.12.005>.

Agrafiotis, D.K.; 2002. “Diversity of Chemical Libraries,” *Encyclopedia of Computational Chemistry*.

Alander, J.T.; Mantere, T.; e Moghadampour, G.; “Testing Software Response Times using a Genetic Algorithm,” *Proceedings of the 3rd Nordic Workshop on Genetic Algorithms and their Applications (3NWGA)* Helsinki, Finlandia, Agosto 1997.

Ali, S.; Briand, L.C.; Hemmati, H.; e Panesar-Walawege, R.K.; 2010. “A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation,” *IEEE Trans. Softw. Eng.* 36, 6 (Novembro 2010), 742-762. DOI=10.1109/TSE.2009.52 <http://dx.doi.org/10.1109/TSE.2009.52>

Alshraideh, M., Bottaci L., e Mahafzah B. A., “Using program data-state scarcity to guide automatic test data generation,” *Software Quality Control* 18, 1, pp 109-144, Março, 2010.

Ammann, P.E.; e Knight, J.K.; 1988. “Data Diversity: An Approach to Software Fault Tolerance,” *IEEE Trans. Comput.* 37, 4 (Abril 1988), 418-425. DOI=10.1109/12.2185 <http://dx.doi.org/10.1109/12.2185>.

Apfelbaum, L.; e Doyle, J. “Model Based Testing,” *Software Quality Week Conference*, pp. 296—300, 1997.

Arcuri, A.; Iqbal, M.Z.; e Briand, L.; “Formal analysis of the effectiveness and predictability of random testing,” In *Proceedings of the 19th international symposium on Software testing and analysis (ISSTA '10)*. ACM, New York, NY, USA, 219-230. 2010.

Andrews, J.; Briand, L.; e Labiche, Y.; “Is mutation an appropriate tool for testing experiments?,” in: *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, Missouri, Maio, 2005, pp. 402–411.

Avizienis, A.; Kelly, J.P.J.; “Fault Tolerance by Design Diversity: Concepts and Experiments,” *Computer*, Vol 17, Issue 8, 1984, p67-80.

Avizienis, A.; 1985. “The N-Version Approach to Fault-Tolerant Software,” *IEEE Trans. Softw. Eng.* 11, 12 (Dezembro 1985), 1491-1501. DOI=10.1109/TSE.1985.231893 <http://dx.doi.org/10.1109/TSE.1985.231893>

Avizienis, A.; Laprie J.C.; e Randell, B.; “Dependability and Its Threats: A Taxonomy,” *Proceedings of the IFIP 18th World Computer Congress*, Kluwer Academic Publishers, Agosto, 2004. Pp. 91-120.

Argollo, M, Bueno, P.M.S., Crespo, A.N., Jino, M., Rosa, M., e Borba, W. “Uma Experiência de Implantação de Processo de Teste em Pequena Empresa”. *Proc.: V Simpósio Brasileiro de Qualidade de Software*, 2006, Vila Velha, ES.

Ayari, K.; Bouktif, S.; e Antoniol, G.; 2007. “Automatic mutation test input data generation via ant colony,” In *Proceedings of the 9th annual conference on Genetic and evolutionary computation* (GECCO '07). ACM, New York, NY, USA, 1074-1081. DOI=10.1145/1276958.1277172 <http://doi.acm.org/10.1145/1276958.1277172>.

Bach, J.; (2003), “Exploratory Testing Explained,” (at <http://www.satisfice.com/articles/et-article.pdf>).

Bach, J.; “Exploratory Testing Dynamics,” *Int. Conf. on Software Testing, Analysis and Review (STARWest)* 2005.

Barbosa, E.F.; Maldonado, J.C.; Vincenzi, A.; Delamaro, M.; Souza, S.; e Jino, M.; “Introdução ao Teste de Software,” XIV Simpósio Brasileiro de Engenharia de Software”, 2000.

Barbosa, E.F.; Maldonado, J.C.; e Vincenzi, A.M.R.; “Toward the determination of sufficient mutant operators for C,” *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 113–136, Maio 2001.

Barbosa, E.F.; Chaim, M.L.; Vincenzi, A.; Delamaro, M.E.; Jino, M.; e Maldonado, J. C.; “Teste Estrutural,” in: Delamaro M. E., Maldonado J.C., Jino M., (Eds.) “Introdução ao Teste de Software,” Elsevier Editora Ltda., 2007.

Barr, R.S.; Golden, B.L.; Kelly, J.P.; Resende, M.G.C.; e Stewart, W.R.; “Designing and reporting on computational experiments with heuristic methods,” *Journal of Heuristics*, 1995, Vol 1, Num 1, P. 9-32.

Bartlett, P.A.; e Entzeroth, M. (Editores); “Exploiting Chemical Diversity for Drug Discovery,” *RCS Publishing*, 2006.

Basili, V.R.; e Selby, R.W.; 1987. “Comparing the Effectiveness of Software Testing Strategies,” *IEEE Trans. Softw. Eng.* 13, 12 (Dezembro 1987), 1278-1296. DOI=10.1109/TSE.1987.232881 <http://dx.doi.org/10.1109/TSE.1987.232881>.

Basili, V.R.; 1996. “The role of experimentation in software engineering: past, current, and future,” In *Proceedings of the 18th international conference on Software engineering* (ICSE '96). IEEE Computer Society, Washington, DC, USA, 442-449.

Beizer, B.; 1990. “Software Testing Techniques,” *Van Nostrand Reinhold*, New York.

- Berek, C.; e Milstein, C.; “The dynamic nature of the antibody repertoire,” *Immunol.Rev.* 1988; 105: 5–26.
- Bertolino, A.;Marchetti, E.;eMuccini, H.; “Introducing a Reasonably Complete and Coherent Approach for Model-based Testing,” *Electron. Notes Theor. Comput.Sci.* 116 Janeiro, pp. 85-97, 2005.
- Bieman, J. M.; e Schultz, J. L.; “An Empirical Evaluation (and specification) of the All-du-paths Testing Criterion,” *Software Engineering Journal*, Jan, 1992, pp.43-51.
- Binder, R.V.; “Testing Object-Oriented Systems.Model Patterns and Tools,”*Addison Wesley*, 2000.
- Bird, D.L.; e Munoz, C.U.; “Automatic Generation of Random Self-Checking Test Cases,”*IBM System Journal*, Vol. 22, N. 3, P. 229-245, 1983.
- Bishop,P. G.; “The variation of software survival times for different operational input profiles,” *In FTSC-23. Digitest of Papers, the Twenty-Third International Symposium on Fault-Tolerant Computing*, p. 98--107. IEEE Computer Society Press, Junho, 1993.
- Blanco, R.; García-Fanjul, J.; e Tuya, J.; “A First Approach to Test Case Generation for BPEL Compositions of Web Services Using Scatter Search,” *In Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW '09)*. IEEE Computer Society, Washington, DC, 2009.
- Blum, C.; e Roli, A.; “Metaheuristics in combinatorial optimization: Overview and conceptual comparison,” *ACM Comput. Surv.* 35, 3 Setembro, 2003, 268-308.
- Bodík, R.; Gupta, R.; e Soffa, M.L.;1997. “Refining data flow information using infeasible paths,” *SIGSOFT Softw. Eng. Notes* 22, 6, 361-377, Novembro, 1997.
- Bottaci, L.; (2001), “A Genetic Algorithm Fitness Function For Mutation Testing,” *Proceedings of the Seminall-Workshop at the 23rd International Conference on Software Engineering*.
- Boettcher, S.;e Percus, A.G.; “Extremal optimization: Methods derived from Co-Evolution,” in: GECCO-99: *Proceedings of the Genetic and Evolutionary Computation Conference*, Morgan Kaufmann, San Francisco, CA, 1999, pp. 825–832.
- Boettcher, S.; e Percus, A.G.;“Optimization with Extremal Dynamics,” *Phys. Rev. Lett.*86, 5211–5214 (2001).
- Boriah, S.; Chandola, V.; e Kumar, V;“Similarity measures for categorical data: A comparative evaluation,”*In Siam International Conference on Data Mining*, pages 243--254. SIAM, Abril 2008.
- Boyer, R.S., Elspas, B. e Karl, N.L., “Select: A Formal System for Testing and Debugging Programs by Symbolic Execution”, *ACM SigPlan Notices*, Vol. 10, N.6, p.234-245, Junho, 1975.
- Briand, L.C.; e Labiche, Y.; “A UML-Based Approach to System Testing,” *In Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, Springer-Verlag, London, UK, 2001, 194-208.
- Briand, L.C.;Labiche, Y.;e Shousha, M.; 2005. “Stress testing real-time systems with genetic algorithms,” *In Proceedings of the 2005 conference on Genetic and evolutionary computation (GECCO '05)*, Hans-Georg Beyer (Ed.). ACM, New York, NY, USA, 1021-1028. DOI=10.1145/1068009.1068183 <http://doi.acm.org/10.1145/1068009.1068183>.

Bringmann, E.; e Krämer, A.; “Systematic testing of the continuous behavior of automotive systems,” In *Proceedings of the 2006 international workshop on Software engineering for automotive systems* (SEAS '06).ACM, New York, NY, USA, 13-20., 2006.

Bryce, R.C.; Lei, Y.; Kuhn, D.R.; e Kacker, R.; “Chapter 14 Combinatorial Testing,” at: <http://csrc.nist.gov/groups/SNS/acts/documents/bryce-lei-kuhn-kacker.pdf>, 2010.

Budd,T.A.;eAngluin, D.; “Two Notions of Correctness and Their Relation to Testing,” *ActaInformatica*, vol. 18, no. 1, pp. 31–45, Março, 1982.

Budynek, J.; Bonabeau, E.; e Shargel, B.; 2005. “Evolving computer intrusion scripts for vulnerability assessment and log analysis,” In *Proceedings of the 2005 conference on Genetic and evolutionary computation* (GECCO '05), Hans-Georg Beyer (Ed.). ACM, New York, NY, USA, 1905-1912. DOI=10.1145/1068009.1068331 <http://doi.acm.org/10.1145/1068009.1068331>.

Bueno, P.M.S.; e Jino, M.; “Identification of Potentially Infeasible Program Paths by Monitoring the Search for Test Data,” In: *Proc. of 15th IEEE International Conference on Automated Software Engineering* (ASE 2000);New York: IEEE, 2000.

Bueno, P.M.S.; “Geração Automática de Dados e Tratamento de Não Executabilidade no Teste Estrutural de Software,” (*Dissertação de Mestrado - DCA/FEEC/Unicamp*). Campinas - SP: Editora da Unicamp, 1999, v.1. p.190.

Bueno, P.M.S.; e Jino, M.; “Geração Automática de Dados e Tratamento de Não Executabilidade no Teste Estrutural de Software,” In *Anais do XIII Simpósio Brasileiro de Engenharia de Software*, (SBES), Outubro, Florianópolis, SC, p.307-322, 1999.

Bueno, P.M.S.; e Jino, M.; “Automatic test data generation for program paths using genetic algorithms,” In *Proc. of International Conference on Software Engineering and Knowledge Engineering* (SEKE), Buenos Aires, pp.2-9, 2001.

Bueno P.M.S.; e Jino M.; “Automatic test data generation for program paths using genetic algorithms,” in *International Journal of Software Engineering and Knowledge Engineering*, Vol. 12, N. 6, pp. 691-709, Dezembro, 2002.

Bueno, P.M.S.;Crespo, A.N.; e Jino, M.; “Analysis of an Artifact Oriented Test Process Model and of Testing Aspects of CMMI,”In: *7th International Conference on Product Focused Software Process Improvement*,Amsterdam., Lecturer Notes in Computer Science. Berlin: Springer-Verlag, 2006. v. 4034. p. 263-277.

Bueno, P.M.S.; Crespo, A.N.;Salviano, C.F.; e Jino, M.; “Analysis of an Artifact Oriented Test Process Model and of Testing Aspects of ISO/IEC 15504,”*VII JornadasIberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento*, Guayaquil, Equador, 2008, IEEE

Bueno, P.M.S.; Wong, W.E.; e Jino, M.; “Improving random test sets using the diversity orientedtest data generation,” in *Proceedings of the Second International Workshop on Random Testing*, pp. 10-17, Atlanta, Georgia, Novembro 2007.

Bueno, P.M.S.; Wong, W.E.; e Jino, M.; “Automatic test data generation using particle systems,” in *Proceedings of the ACM Symposium of Applied Computing*, pp. 809-814, Fortaleza, Brazil, Março, 2008.

Bueno, P.M.S.;Jino, M.; e Wong, W.E.; “Diversity Oriented Test Data Generation Using Metaheuristic Search Techniques,” *Information Sciences*, Elsevier, (doi:10.1016/j.ins.2011.01.025), 2011.

Bueno, P.M.S.; “SBST aplicado ao teste estrutural,” In: Vergilio, S.R.; Bueno, P.M.S.; Dias Neto, A. C.; Martins E.;Vincenzi A., “Tutorial: Geração de Dados de Teste: Principais Conceitos e Técnicas,” *Anais do II Congresso Brasileiro de Software: Teoria e Prática*, 2011, São Paulo.

Canfora, G.; Di Penta, M.; Esposito, R.; e Villani, M.L.; 2005. “An approach for QoS-aware service composition based on genetic algorithms,” In *Proceedings of the 2005 conference on Genetic and evolutionary computation* (GECCO '05), Hans-Georg Beyer (Ed.). ACM, New York, NY, USA, 1069-1075. DOI=10.1145/1068009.1068189 <http://doi.acm.org/10.1145/1068009.1068189>.

Carniello, A.;Jino, M.; e Chaim, M.L.; “StructuralTestingwith Use Cases,” *Anais do WER04 – Workshop em Engenharia de Requisitos*, Argentina, Dezembro, 2004. PP. 140-151, 2004.

Caflisch, R.E.; (1998). “Monte Carlo andquasi-Monte Carlo methods,” *Acta Numerica*. 7. Cambridge University Press. pp. 1–49.

Chaim, M.L.; “POKE-TOOL – Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseados Em Análise de Fluxo de Dados,” *Dissertação de Mestrado, DCA/FEE/UNICAMP*, Campinas, SP, Brasil, 1991.

Chaim, M.L.; Maldonado, J.C.;Jino, M.; Vilela, P.; e Bueno P.M.S.; “POKE-TOOL - Manual do Usuário,” *DCA/FEEC/UNICAMP*, Campinas, SP, Brasil, 1995.

Chen, M.H.;Lyu, M.R.;e Wong,W.E.;“Effect of code coverage on software reliability measurement,” *IEEE Transactions on Reliability*, 50(2):165--170, Junho 2001.

Chen, L.;May, J.; e Hughes, G.; “Estimation of software diversity by fault simulation and failure searching,” (ISSRE 2001).*Proc. 12th International Symposium on Software Reliability Engineering*, 2001.P. 122-131, IEEE.

Chen,T.Y.;e Yu,Y.T.;1994. “On the Relationship Between Partition and Random Testing,” *IEEE Trans. Softw. Eng.* 20, 12 (Dezembro 1994), 977-980. DOI=10.1109/32.368132 <http://dx.doi.org/10.1109/32.368132>.

Chen, T.Y.;e Kuo,F.C.; 2006.“Is adaptive random testing really better than random testing,” In *Proceedings of the 1st international workshop on Random testing* (RT '06). ACM, New York, NY, USA, 64-69.DOI=10.1145/1145735.1145745 <http://doi.acm.org/10.1145/1145735.1145745>.

Chen, T.Y.; Huang, D.H.; e Zhou. Z.Q.; 2006. “Adaptive random testing through iterative partitioning,” In *Proceedings of the 11th Ada-Europe international conference on Reliable Software Technologies* (Ada-Europe'06), Springer-Verlag, Berlin, Heidelberg, 155-166.DOI=10.1007/11767077_13 http://dx.doi.org/10.1007/11767077_13.

Chen, T.Y.; e Merkel, R.; 2007. “Quasi-random testing,” *In Transactions on Reliability*, 2007, Issue 3, Vol 56, p. 562-568.

Chen, T.;Leung, H.; Mak, I.; “Adaptive Random Testing,” *Advances in Computer Science - ASIAN 2004.Higher-Level Decision Making – Lecture Notes in Computer Science*, Vol. 3321, pp. 320-329. Springer Berlin / Heidelberg, 2005.

Chillarege, R.; "Software Testing Best Practices," *IBM Research Report RC 21457, Log 96856*. IBM Research, York Town Heights, 1999.

Clarke, L.A.; "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, Vol. SE-2, n 3, p.215-222, Setembro, 1976.

Clarke, L.A.; e Richardson, D.J.; 1983. "The application of error-sensitive testing strategies to debugging," In *Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on High-level debugging* (SIGSOFT '83). ACM, New York, NY, USA, 45-52. DOI=10.1145/800007.808007 <http://doi.acm.org/10.1145/800007.808007>

Clarke, J.M.; "Automated test generation from a behavioral model," *Proceedings of the 11th International Software Quality Week (QW 98)*, Maio 1998.

Chow, T.S.; "Testing Software Design Modeled by Finite-State Machines," *IEEE Trans. Softw. Eng.* 4, 3 (Maio 1978), 178-187, 1978.

Clarke, L.; Podgurski, A.; Richardson, D.; e Seil, S.; "A Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Trans. Soft. Eng.*, Vol. 15, No. 11, pp. 244-251, Nov. 1989.

Clarke, L.A.; Hassell, J.; Richardson, D.J.; "A Close Look at Domain Testing," *IEEE - Trans. Software Eng.*, vol. SE - 8, No.4, pp. 380-390, Jul, 1982.

Cohen D.M.; Dalal S.R.; Fredman M.L.; e Patton G.C.; "The AETG system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, 23(7), 1997.

Cohen, M.B.; Gibbons, P.B.; Mugridge, W.B.; e Colbourn, C.J.; "Constructing test suites for interaction testing," In *Proceedings of the 25th International Conference on Software Engineering* (ICSE '03). IEEE Computer Society, Washington, DC, USA, 38-48, 2003.

Conway, J. H.; Goodman-Strauss C.; e Sloane, N.J.A.; "Recent progress in sphere packing," *Current Developments in Mathematics* (Cambridge, MA, 1999), B. Mazur, W. Schmid, S. T. Yau, D. Jerison, I. Singer and D. Stroock (editores), International Press, Somerville, MA, 1999, pp. 37-76.

Crespo, A.N.; Jino, M.; Pasquini, A.; e Maldonado, J.C.; 2008. "A binomial software reliability model based on coverage of structural testing criteria," *Empirical Softw. Eng.* 13, 2 (Abril 2008), 185-209. DOI=10.1007/s10664-007-9055-3 <http://dx.doi.org/10.1007/s10664-007-9055-3>.

Croarken, M.; 2003. "Tabulating the Heavens: Computing the Nautical Almanac in 18th-Century England," *IEEE Ann. Hist. Comput.* 25, 3 (Julho 2003), 48-61. DOI=10.1109/MAHC.2003.1226655 <http://dx.doi.org/10.1109/MAHC.2003.1226655>.

Dalal, S.R.; Jain, A.; Karunanithi, N.; Leaton, J.M.; Lott, C.M.; Patton, G.C.; e Horowitz, B.M.; "Model-based testing in practice," In *Proceedings of the 21st international conference on Software engineering* (ICSE '99). ACM, New York, NY, USA, 285-294. 1999.

Darwin, C.; "On the Origin of Species," 1872. (at: The Talk Origins Archive: <http://www.talkorigins.org/faqs/origin.html>).

Darwin, C.; "A Origem das Espécies, no meio da seleção natural ou a luta pela existência na natureza," 1 vol. *E-book, versão traduzida em português*, LELLO & IRMÃO – EDITORES, 2003. (at: ecologia.ib.usp.br/ffa/arquivos/abril/darwin1.pdf). Consultado em Novembro 2011.

De Castro, L.N.; “Engenharia Imunológica: Desenvolvimento e Aplicação de Ferramentas Computacionais Inspiradas em Sistemas Imunológicos Artificiais,” *Tese de Doutorado, DCA/FEEC/Unicamp*, 2001.

De Castro, L.N.; e Von Zuben, F.J.; (2001), "An Immunological Approach to Initialize Feedforward Neural Network Weights," *Proceedings do ICANNGA 2001 (International Conference on Artificial Neural Networks and Genetic Algorithms)*, pp. 126-129.

De Castro, L.N.; e Timmis, J.; (2002). “Artificial Immune Systems: A New Computational Intelligence Approach,” *Springer*.

De Castro; L.N.; e Von Zuben F.J.; “Learning and optimization using the clonal selection principle,” *IEEE Trans. on Evol. Comp.*, vol 6, no 3, pp. 239--251, (2002).

De Jong, K.A.; (1975). “An Analysis of the Behavior of a Class of Genetic Adaptive Systems,” *Unpublished doctoral dissertation. University of Michigan*, Ann Arbor, MI.

De Masi, D.; “Criatividade e Grupos Criativos,” Ed. *Sextante*, 2005.

De Sousa, F.L.; Ramos, F.M.; Paglione, P.; e Ramos, F.M.; “New Stochastic Algorithm for Design Optimization,” *AIAA Journal*. Vol. 41, Iss. 9, *American Institute of Aeronautics and Astronautics*, p.: 1808-1818, 2006.

Delamaro, M.E. e Maldonado, J.C.; “A tool for the assessment of test adequacy for C programs,” in *Proceedings of the Conference on Performability in Computing System*, pp. 79-95, New Jersey, Julho, 1996.

Delamaro, M.E.; Maldonado, J.C.; e Mathur, A.P.; “Interface Mutation: An Approach for Integration Testing,” *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, Maio 2001.

Delamaro, M.E.; Maldonado J.C.; e Jino M.; (Eds.) “Introdução ao Teste de Software,” *Elsevier Editora Ltda.*, 2007.

Delamaro, M.E.; Barbosa, E.F.; Vincenzi, A.; e Maldonado, J.C.; “Teste de Mutação,” in: Delamaro M. E., Maldonado J.C., Jino M., (Eds.) “Introdução ao Teste de Software,” *Elsevier Editora Ltda.*, 2007.

DeMillo, R.A.; e Offutt, A. J.; “Experimental Results from an Automatic Test Case Generator,” *ACM Trans. Soft. Eng. Methodology*, Vol. 2, No. 2, pp. 109-127, Apr. 1993.

DeMillo, R.A.; e Offutt, A.J.; “Constraint Based Automatic Test Data Generation,” *IEEE - Trans. Software Eng.* vol. 17, No. 9, pp. 900-910, Setembro, 1991.

DeMillo, R.A.; Lipton, R.J.; e Sayward, F.G.; “Hints on Test Data Selection: Help for a Practicing Programmer,” *IEEE - Computer*, pp. 34-41, Abril, 1978.

DeMillo, R.A.; Mathur, A.P.; e Wong, E.W.; “Some Critical Remarks on a Hierarchy of Fault-Detecting Abilities of Test Methods,” *IEEE - Trans. Software Eng.* vol. 21, No. 10, pp. 858-863, Outubro, 1995.

Derderian, K.; Hierons, R.M.; Harman, M.; e Guo, Q.; “Generating feasible input sequences for extended finite state machines (EFSMs) using genetic algorithms,” In *Proceedings of the 2005 conference on Genetic and evolutionary computation (GECCO '05)*, ACM, New York, NY, USA, 1081-1082. 2005.

- Di Penta, M.; Canfora, G.; Esposito, G.; Mazza, V. e Bruno. M.; 2007. "Search-based testing of service level agreements," In *Proceedings of the 9th annual conference on Genetic and evolutionary computation* (GECCO '07). ACM, New York, NY, USA, 1090-1097. DOI=10.1145/1276958.1277174 <http://doi.acm.org/10.1145/1276958.1277174>,
- Dias Neto, A.C.; Subramanyan, R.; Vieira, M.; e Travassos, G.H.; "A survey on model-based testing approaches: a systematic review," In *Proceedings of the 1st ACM international workshop on empirical assessment of software engineering languages and technologies 2007* (WEASEL Tech '07). ACM, New York, NY, USA, 2007.
- Dickinson, W.; Leon, D.; e Podgurski, A.; "Finding failures by cluster analysis of execution profiles," In *Proceedings of the 23rd International Conference on Software Engineering* (ICSE '01). IEEE Computer Society, Washington, DC, USA, 339-348, 2001.
- Dijkstra E.; "Notes on Structured Programming," in Dahl O, Dijkstra E, Hoare C (Eds.) *Structured Programming*, (Academic Press 1972).
- Dorigo, M.; e Di Caro, G.; 1999. "The ant colony optimization meta-heuristic," In *New ideas in optimization*, David Corne, Marco Dorigo, Fred Glover, Dipankar Dasgupta, Pablo Moscato, Riccardo Poli, and Kenneth V. Price (Eds.). McGraw-Hill Ltd., UK, Maidenhead, UK, England 11-32.
- Droste, S.; e Wiesmann, D.; 2003. "On the design of problem-specific evolutionary algorithms," In *Advances in evolutionary computing*, Ashish Ghosh and Shigeyoshi Tsutsui (Eds.). Springer-Verlag New York, Inc., New York, NY, USA 153-173.
- Dunham, J.R.; e Finelli, G.B.; "Real-time software failure characterization," *Aerospace and Electronic Systems Magazine*, IEEE, Novembro, 1990, p. 38-44.
- Duran, J. W.; e Ntafos, S. C.; "An Evaluation of Random Testing," *IEEE Trans. Software*, Vol. SE-10, No. 7, pp. 438-444, Julho 1984.
- Dyba T.; Kitchenham, B.A.; e Jorgensen, M.; 2005. "Evidence-Based Software Engineering for Practitioners," *IEEE Softw.* 22, 1 (Janeiro 2005), 58-65. DOI=10.1109/MS.2005.6 <http://dx.doi.org/10.1109/MS.2005.6>.
- Eberhart, R.C.; e Shi, Y.; "Particle swarm optimization: developments, applications and resources," *Proc. congress on evolutionary computation 2001* IEEE service center, Piscataway, NJ., Seoul, Korea., 2001.
- Edvardsson, J.; "A survey on automatic test data generation," In *Proceedings of the Second Conference on Computer Science and Engineering* in Linköping, pages 21-28. ECSEL, Outubro 1999.
- Eglese, R.W.; 1990. "Simulated annealing: A tool for operational research," *European Journal of Operational Research*, Elsevier, vol. 46(3), pages 271-281, Junho.
- Erdogmus, H.; Morisio, M.; e Torchiano, M.; "On the effectiveness of the test-first approach to programming," *IEEE Trans. Software*, Vol. 31-3, pp. 226 – 237, 2005.
- Ehrlich, W.K.; Iannino, A.; Prasanna, B.S.; Stampfel, J.P.; e Wu, J.R.; "How faults cause software failures: implications for software reliability engineering," *Second International Symposium on*

Software Reliability Engineering, pp. 233–241 (Austin, TX, Maio 17–18, 1991). Los Alamitos, CA: IEEE Computer Society Press, 1991.

Fabbri, S.C.P.F.; Maldonado, J.C.; Masiero, P.C.; e Delamaro M.E.; “Proteum/FSM: A Tool to Support Finite State Machine Validation Based on Mutation Testing,” In *Proceedings of the 19th International Conference of the Chilean Computer Science Society (SCCC '99)*. IEEE Computer Society, Washington, DC, USA. 1999.

Fabbri, S.C.P.F.; Maldonado, J.C.; Masiero, P.C.; e Sugeta T.; Mutation Testing Applied to Validate Specifications Based on Statecharts,” in *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE'99)*, Boca Raton, Florida, 1-4 Novembro, 1999, p. 210.

Fantinato, M.; e Jino, M.; “Applying Extended Finite State Machines in Software Testing of Interactive Systems,” *Proceedings of the 10th International Workshop on Design, Specification, and Verification of Interactive Systems*, Funchal, Lecture Notes in Computer Science, v. 2844, pp. 109-131, 2003.

El-Far, I. K.; e Whittaker, J.A.; “Model-Based Software Testing,” *Encyclopedia of Software Engineering* (edited by J. J. Marciniak). Wiley. 2001.

Feldt, R.; Torkar, R.; Gorschek, T.; e Afzal, W.; 2008. “Searching for Cognitively Diverse Tests: Towards Universal Test Diversity Metrics,” In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW '08)*. IEEE Computer Society, Washington, DC, USA, 178-186. DOI=10.1109/ICSTW.2008.36 <http://dx.doi.org/10.1109/ICSTW.2008.36>.

Ferguson R.; e Korel, B.; “The chaining approach for software test data generation,” *ACM Trans. Softw. Eng. Methodol.* 5, 1 63-86, Janeiro, 1996.

Fischer, G.; 2005. “Distances and diversity: sources for social creativity,” In *Proceedings of the 5th conference on Creativity & cognition (C&C '05)*. ACM, New York, NY, USA, 128-136. DOI=10.1145/1056224.1056243 <http://doi.acm.org/10.1145/1056224.1056243>.

Forgács, I.; e Bertolino, A.; 1997. Feasible test path selection by principal slicing. *SIGSOFT Softw. Eng. Notes* 22, 6, 378-394. Novembro, 1997.

Forrester, J.E.; e Miller, B.P.; “An empirical study of the robustness of Windows NT applications using random testing,” In *4th USENIX Windows System Symposium*, pp 59–68, Seattle, WA, USA, Agosto, 2000.

Frankl, P.G.; e Weyuker, E.J.; “An Applicable Family of Data Flow Testing Criteria,” *IEEE Trans on Software Eng.*, Vol. 14, No. 10, pp. 1483-1498, Outubro, 1988.

Frankl, P.G.; e Weyuker, E.L.; “A Formal Analysis of the Fault-Detecting Ability of Testing Methods,” *IEEE Trans. Software Eng.*, Vol. 19, No. 3, pp. 203-213, Março 1993.

Frankl, P.G.; e Weyuker, E.L.; 1993. “Provable Improvements on Branch Testing,” *IEEE Trans. Softw. Eng.* 19, 10 (October 1993), 962-975. DOI=10.1109/32.245738 <http://dx.doi.org/10.1109/32.245738>.

Frankl, P.G.; e Weiss, S.N.; “An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing,” *IEEE Trans. Software Eng.*, Vol. 19, No. 9, pp. 774-786, Agosto 1993.

Frankl, P.G.; e Weyuker, E.L.; “Provable Improvements on Branch Testing,” *IEEE Trans. Software Eng.*, Vol. 19, No. 10, pp. 962-974, Outubro, 1993.

Frankl, P.; Hamlet, D.; Littlewood, B.; e Strigini, L.; 1997; “Choosing a testing method to deliver reliability,” In *Proceedings of the 19th international conference on Software engineering (ICSE '97)*. ACM, New York, NY, USA, 68-78. DOI=10.1145/253228.253244 <http://doi.acm.org/10.1145/253228.253244>.

Finelli, G.B.; “NASA Software Failure Characterization Experiments,” *Reliab. Eng. & Syst. Saf.*, vol. 32, 1991, pp. 155-169.

Fröhlich, P; e Link, J.; “Automated Test Case Generation from Dynamic Models,” In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP '00)*, Springer-Verlag, London, UK, 472-492, 2000.

Fujisawa, K. e Takefuji, Y.; “A balloon net discovering improved solutions in one of unsolved problems in geometry: a problem of spreading points in a unit square,” in: *Proceedings of IEEE International Conference on Neural Networks*, Perth, WA, Australia, Dezembro 1995, pp. 2208–2210.

Gallagher, M.J.; e Narasimhan, V.N.; “ADTEST: A Test Data Generation Suite for Ada Software Systems,” *IEEE Trans. Softw. Eng.* 23, 8, 473-484, Agosto, 1997.

Ghani, K.; Clark, J.A.; e Zhan, Y.; “Comparing algorithms for search-based test data generation of Matlab Simulink models,” In *Proceedings of the Eleventh conference on Congress on Evolutionary Computation (CEC'09)*. IEEE Press, Piscataway, NJ, USA, 2940-2947, 2009.

Ghazi, S.A.; e Ahmed, M.; “Pair-wise test coverage using genetic algorithms,” In *Proceedings of the Congress on Evolutionary Computation (CEC'03)*. IEEE Computer Society, Los Alamitos, CA, 1420–1424. 2003.

Glover, F.; Laguna, M.; e Martí, R.; (2000). “Fundamentals of Scatter Search and Path Relinking,” *Control and Cybernetics* 39(3), 653-684.

Goodenough, J.B.; e Gerhart, S.L.; “Toward a theory of test data selection,” *IEEE Trans. Softw. Eng. SE-3* (Junho), 1975.

Gotlieb, A.; Botella, B.; e Rueher, M.; “Automatic test data generation using constraint solving techniques,” *SIGSOFT Softw. Eng. Notes* 23, 2, 53-62, Março 1998.

Gotlieb, A.; e Petit, M.; 2006. “Path-oriented random testing,” In *Proceedings of the 1st international workshop on Random testing (RT '06)*. ACM, New York, NY, USA, 28-35.

Goldberg, D.E.; “Genetic algorithms in search, optimization, and machine learning,” *Addison-Wesley*, 1989.

Goldberg, M.; “The Packing of Equal Circles in a Square,” *Mathematics Magazine* Vol. 43, No. 1 (Jan., 1970), pp. 24-30.

Grindal, M.; Offutt, J.; e Andler, S.F.; “Combination testing strategies: A survey,” *Software Testing, Verification, and Reliability*, Wiley, Vol. 15, 167-199, 2005.

Gupta, N.; Mathur, A.P.; e Soffa, M.L.; “Automated test data generation using an iterative relaxation method,” *SIGSOFT Softw. Eng. Notes* 23, 6, 231-244. Novembro, 1998.

- Gutjahr, W.J.; 1999. "Partition Testing vs. Random Testing: The Influence of Uncertainty," *IEEE Trans. Softw. Eng.* 25, 5 (Setembro 1999), 661-674. DOI=10.1109/32.815325 <http://dx.doi.org/10.1109/32.815325>.
- Hales, T.C.; 1992. "The sphere packing problem," *J. Comput. Appl. Math.* 44, 1 (December 1992), 41-76. DOI=10.1016/0377-0427(92)90052-Y [http://dx.doi.org/10.1016/0377-0427\(92\)90052-Y](http://dx.doi.org/10.1016/0377-0427(92)90052-Y).
- Halliday, D.; Resnick, R.; e Walker, J.; "Fundamentos de Física," 8a Ed., LTC Editora, 2009.
- Hamlet, D.; e Taylor, R.; "Partition testing does not inspire confidence," *IEEE Trans. Softw. Eng.* 16 (Dez.), 206-215. 1990.
- Hamlet, D.; "Connecting test coverage to software dependability," *Proc. 5th Intl. Symp. on Soft. Rel.* 1994, p. 158-165, IEEE.
- Hamlet, D.; 2002. "Continuity in software systems," *SIGSOFT Softw. Eng. Notes* 27, 4 (Julho 2002), 196-200. DOI=10.1145/566171.566203 <http://doi.acm.org/10.1145/566171.566203>.
- Hamlet, D.; "When Only Random Testing will do," In *Proceedings of the First International Workshop on Random Testing*, Jul 20, 2006, Portland, ME, USA.
- Hamlet, R.; "Random testing," In J. Marciniak, editor, *Encyclopedia of Soft. Eng.* Wiley, 2nd edition, 2005.
- Harman, M.; e Jones, B.F.; "Search-based software engineering," *Information & Software Technology*, Vol. 43, No. 14, pp. 833-839 (2001).
- Harman, M.; 2007. "The Current State and Future of Search Based Software Engineering," In *2007 Future of Software Engineering (FOSE '07)*. IEEE Computer Society, Washington, DC, USA, 342-357. DOI=10.1109/FOSE.2007.29 <http://dx.doi.org/10.1109/FOSE.2007.29>.
- Harman, M.; Mansouri, A.; e Zhang, Y.; "Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications," *Department of Computer Science*, King's College London, TR-09-03, Abril, 2009.
- Harrold, M. J.; e Soffa, M. L.; "Selecting and Using Data for Integration Testing," *IEEE Software*, Vol. 8, No. 2, Março 1991, pp. 58-65.
- Hartman, A.; e Nagin, K.; "The AGEDIS tools for model based testing," *SIGSOFT Softw. Eng. Notes* 29, 4 (Julho 2004), 129-132., 2004.
- Hedley, D.; e Hennell, M.A.; 1985. "The causes and effects of infeasible paths in computer programs," In *Proceedings of the 8th international conference on Software engineering (ICSE '85)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 259-266.
- Hemmati, H.; Briand, L.; Arcuri, A.; e Ali, S.; "An enhanced test case selection approach for model-based testing: an industrial case study," In *Proceedings of the eighteenth ACM SIGSOFT international symposium on foundations of software engineering (FSE '10)*. ACM, New York, NY, USA, 267-276. 2010.
- Herman, P.M.; "A Data Flow Analysis Approach to Program Testing," *The Australian Computer Journal*, Vol. 8, No. 3, pp. 92-96, Nov. 1976.
- Hetzl, B.; 1988, "The Complete Guide to Software Testing (2nd Ed.)," QED Information Sciences, Inc.

- Hightower, R.R.; Forrest, S.; e Perelson, A.S.; 1995. "The Evolution of Emergent Organization in Immune System Gene Libraries," In *Proceedings of the 6th International Conference on Genetic Algorithms*, Larry J. Eshelman (Ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 344-350.
- Hockney, R.W.; e Eastwood, J.W.; 1988. *"Computer Simulation Using Particles,"* Taylor & Francis, Inc., Bristol, PA, USA.
- Holland, J.H.; 1992. *"Adaptation in Natural and Artificial Systems,"* MIT Press, Cambridge, MA, USA.
- Horgan, J.R.; London, S.; e Lyu, M.R.; 1994. "Achieving software quality with testing coverage measures," *Computer* 27, 9 (Setembro 1994), 60-69. DOI=10.1109/2.312032 <http://dx.doi.org/10.1109/2.312032>.
- Hoskins, D.; Turban, R.C.; e Colbourn, C.J.; "Experimental designs in software engineering: d-optimal designs and covering arrays," In *Proceedings of the 2004 ACM workshop on Interdisciplinary software engineering research (WISER '04)*. ACM, New York, NY, USA, 55-66, 2004.
- Howden, W.E.; 1975. "Methodology for the Generation of Program Test Data," *IEEE Trans. Comput.* 24, 5 (Maio 1975), 554-560. DOI=10.1109/T-C.1975.224259 <http://dx.doi.org/10.1109/T-C.1975.224259>.
- Howden, W.E.; "Reliability of the Path Analysis Testing Strategy," *IEEE Trans. Soft.Eng.* Vol. SE-2, No. 3, pp. 208-215, 1976.
- Howden, W.E.; "Symbolic testing and Dissect Symbolic Evaluation System," *IEEE Transactions on Software Engineering*, Vol. SE-3, N. 4, P. 266-278, Julho, 1977.
- Howden, W.E.; "Weak Mutation Testing and Completeness of Test Sets," *IEEE Trans. Soft.Eng.* Vol. SE-8, No. 4, pp. 371-379, 1982.
- IEEE, "IEEE Std 829 Standard for Software Test Documentation," New York, 2008.
- IEEE, "IEEE Std 610. standard glossary of software engineering terminology," 1990.
- ISO/IEC, "ISO/IEC 9126-1: Software engineering -- Product Quality -- Part 1 Quality model," 2001.
- ISO/IEC, "ISO/IEC 15504-5 Information Technology — Process Assessment — Part 5: An exemplar Process Assessment Model," The International Organization for Standardization and the International Electrotechnical Commission, 2006.
- ISO/IEC, "ISO/IEC 29119 Software Testing, part 1 (definitions and concepts)," 2010.
- Jacobson, I.; Booch, G.; Rumbaugh, J.; "The Unified Software Development Process," Addison Wesley, 1999.
- Jedlitschka, A.; Ciolkowski, M.; e Pfahl, D.; (2008), "Reporting Experiments in Software Engineering," In: *Guide to Advanced Empirical Software Engineering*, 2008, Springer-Verlag New York, Inc., Secaucus, NJ, USA, pp. 201-228.

Jia, Y.; e Harman, M.; "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, 10 Jun. 2010.IEEE computer Society Digital Library.IEEE Computer Society.

Jones, F.;Sthamer,H.H.;eEyres, D.E.;“Automatic structural testing using genetic algorithms,” *Software Engineering Journal*,Setembro, p,299-306, 1996.

Kalaji, A.S.;Hierons,R.M.; e Swift, S.;“A Search-Based Approach for Automatic Test Generation from Extended Finite State Machine (EFSM),” *Testing: Academia & Industry Conference - Practice and Research Techniques (TAIC-PART)*. 2009.

Kaner, C.; Bach, J.; e Pettichord, B.; “Lessons Learned in Software Testing: A Context-Driven Approach,”*Willey Computer Publishing*, 2002.

Kansomkeat S.; e Rivepiboon, W.; “Automated-generating test case using UML statechart diagrams,” *In Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology (SAICSIT '03)*, 2003.

Kayacik, H.G.;Zincir-Heywood, A.N.; e Heywood, M.; “Evolving successful stack overflow attacks for vulnerability testing,” *21st Annual Computer Security Applications Conference*, 21st Annual, Dezembro, 2005;

Kennedy, J.; e Eberhart, R.; (1995).“Particle Swarm Optimization,”*Proceedings of IEEE International Conference on Neural Networks.IV*. pp. 1942–1948. doi:10.1109/ICNN.1995.488968.

Kennedy, J.; e Eberhart, R.C.; (2001).“Swarm Intelligence,”*MorganKaufmann*.

Kirkpatrick, S.; Gerlatt, C.D.Jr.; e Vecchi, M.P.; (1983); “Optimization by Simulated Annealing,” *Science*, 220, 671-680.

Kirkpatrick, S.; (1984), “Optimization by Simulated Annealing - Quantitative Studies,” *J. Stat. Phys.*, 34, pp. 975-986.

Kitchenham, B.A.; Dyba, T.; e Jorgensen, M.; 2004. “Evidence-Based Software Engineering,” *In Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 273-281.

Klaib, M.F.J.;Zamli, K.Z.; Isa, N.A.M.;Younis, M.I.; e Abdullah, R.; “G2Way A Backtracking Strategy for Pairwise Test Data Generation,” *In Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference (APSEC '08)*.IEEE Computer Society, Washington, DC, USA, 463-470, 2008.

Knight, J.C.; e Leveson, N.G.; “An experimental evaluation of the assumption of independence in multiversionprogramming,” *IEEE Transactions on Software Engineering*,1986, vol. 12, no1, pp. 96-109.

Korel,B.;1990.“Automated Software Test Data Generation,”*IEEE Trans. Softw. Eng.* 16, 8 (Agosto 1990), 870-879.

Korel,B.;“Automated test data generation for programs with procedures,” *In Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '96)*, ACM, New York, NY, USA, 209-215. 1996.

- Krishnan, R.; Krishna, S.M.; e Nandhan, P.S.; “Combinatorial testing: learnings from our experience,” *SIGSOFT Softw. Eng. Notes* 32, 3 (Maio 2007), 1-8, 2007.
- Kuhn, D.R.; Kacker, R.; Lei, Y.; e Hunter, J.; “Combinatorial Software Testing,” *Computer* 42, 8 (Agosto 2009), 94-96. DOI=10.1109/MC.2009.253. 2009.
- Kuhn, D.R.; Wallace, D.R.; e Gallo, A.; “Software Fault Interactions and Implications for Software Testing,” *IEEE Transactions on Software Engineering*, 30(6): 418-421, 2004.
- Lakhotia, K.; Harman, M.; e McMinn, P.; 2007. “A multi-objective approach to search-based test data generation,” In *Proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO '07)*. ACM, New York, NY, USA, 1098-1105. DOI=10.1145/1276958.1277175 <http://doi.acm.org/10.1145/1276958.1277175>.
- Larson, R.; e Farber, B.; “Elementary Statistics: Picturing the World,” fourth ed., *PrenticeHall*, 2009.
- Laski, J.; e Korel, B.; “A Data Flow Oriented Program Testing Strategy,” *IEEE Trans. Software Eng.*, Vol. SE-9, No.3, pp. 347-354, Maio, 1983.
- Lei, Y.; Kacker, R.; Kuhn, D.R.; Okun, V.; e Lawrence, J.; “Ipog/ipog-d: Efficient test generation for multi-way combinatorial testing,” *Softw. Test., Verif. Reliab.* 18, 3, 125–148. 2008.
- Leon, D.; e Podgurski, A.; “A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases,” In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE '03)*. IEEE Computer Society, Washington, DC, USA, 2003.
- Liang, J.J.; Qin, A.K.; Suganthan, P.N.; e Baskar, S.; “Comprehensive learning particle swarm optimizer for global optimization of multimodal functions,” *IEEE Trans. Evolutionary Computation*, 10(3): p.281-295, 2006.
- Lin, J.; e Yeh, P.; “Automatic test data generation for path testing using GAs,” *Information Sciences*, Volume 131, Issue 1-4, pp.47-64, Jan. 2001.
- Linnenkugel, U.; e Müllerburg, M.; “Test Data Selection Criteria for (Software) Integration Testing,” in *Proceedings of the First International Conference on Systems Integration*. Monstown, New Jersey, pp.709-717, Abril 1990.
- Lions, J.L.; “ARIANE 5, Flight 501 Failure, Report by the Inquiry Board,” *European Space Agency*, Julho 1996.
- Littlewood, B.; Popov, P.T.; Strigini, L.; e Shryane, N.; 2000. “Modeling the Effects of Combining Diverse Software Fault Detection Techniques,” *IEEE Trans. Softw. Eng.* 26, 12 (Dezembro 2000), 1157-1167. DOI=10.1109/32.888629 <http://dx.doi.org/10.1109/32.888629>.
- Littlewood, B.; e Strigini, L.; 2000. “Software reliability and dependability: a roadmap,” In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*. ACM, New York, NY, USA, 175-188. DOI=10.1145/336512.336551 <http://doi.acm.org/10.1145/336512.336551>.
- Littlewood, B.; Popov, P.T.; Strigini, L.; 2001. “Modeling software design diversity: a review,” *ACM Comput. Surv.* 33, 2 (Junho 2001), 177-208. DOI=10.1145/384.

Londesbrough, I.; "A Test Process for all Lifecycles," *IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)* 2008.

Lutz, M.; "Testing Tools," *IEEE Software*, Maio 1990, pp. 53-57.

Lyu, M.R.; Chen, J.; e Avizienis, A.; "Software Diversity Metrics and Measurements," *In proceedings of the Sixteenth Annual International Computer Software and Applications Conference (COMPSAC '92)*, Chigaco, IL USA. 1992. pp. 69-78.

Lyu, M.R.; e He, Y.; "Improving the N-Version Programming Process Through the Evolution of a Design Paradigm," *IEEE Transactions on Reliability*. 1993, Vol. 42, 2, pp. 179-189.

Malaiya, Y.K.; "Antirandom Testing: Getting the most out of black-box testing," in *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, pp. 86-95, Toulouse, France, Outubro 1995.

Malaiya, Y.K.; Li, M.N.; Bieman, J.M.; e Karcich, R.; "Software reliability growth with test coverage," *IEEE Transactions on Reliability*, 51(4):420-426, Dezembro 2002.

Maldonado, J.C.; "Critérios Potenciais Usos: uma Contribuição ao Teste Estrutural de Software," Tese de Doutorado, *DCA/FEE/UNICAMP*, Campinas, 1991.

Maldonado, J.C.; Chaim, M.L.; e Jino, M.; "Bridging the Gap in the Presence of Infeasible Paths: Potential Uses Testing Criteria," *II International Conference of the SCCC*, pp. 323-339, 1992.

Maldonado, J.C.; Delamaro, M.E.; Fabbri, S.C.P.F.A.S.; Simao, A.; Vicenzi, A.; e Masiero, P.C.; "Proteum - a Family of Tools to Support Specification and Program Testing Based on Mutation," *In IEEE Reliability Society, Mutation 2000 Symposium*, San José, 2000, pp. 146-149.

Malevris, N.; Yates, D.F.; e Veevers, A.; 1990. "Predictive metric for likely feasibility of program paths," *J. Electron. Mater.* 19, 6 (Junho 1990), 115-118.

Mantere, T.; e Alander, J.T.; 2005. "Evolutionary software engineering, a review," *Appl. Soft Comput.* 5, 3 (Março 2005), 315-331. DOI=10.1016/j.asoc.2004.08.004 <http://dx.doi.org/10.1016/j.asoc.2004.08.004>.

Martí, R.; Laguna, M. e Glover, F.; "Principles of Scatter Search," *European Journal of Operational Research*, Vol. 169, Issue 2, 1, Março, 2006, P. 359-372.

Mathur, A.P.; "Performance, Effectiveness, and Reliability Issues in Software Testing," in *Proceedings of the 5th International Computer Software and Applications Conference (COMPSAC)*, Tokyo, Japan, 11-13, Setembro 1991, pp. 604-605.

Mayer, J.; 2005. "Lattice-based adaptive random testing," *In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05)*. ACM, New York, NY, USA, 333-336. DOI=10.1145/1101908.1101963 <http://doi.acm.org/10.1145/1101908.1101963>.

Mayer, J.; 2006. "Adaptive random testing with randomly translated failure region," *In Proceedings of the 1st international workshop on Random testing (RT '06)*. ACM, New York, NY, USA, 70-77. DOI=10.1145/1145735.1145746 <http://doi.acm.org/10.1145/1145735.1145746>.

McCabe, T., "A complexity measure," *IEEE Trans. Software Eng.*, 2 4 (Apr. 1976), 308-320.

- McMinn, P.; 2004. "Search-based software test data generation: a survey," *Research Articles. Softw. Test. Verif. Reliab.* 14, 2 (Junho 2004), 105-156. DOI=10.1002/stvr.v14:2 <http://dx.doi.org/10.1002/stvr.v14:2>
- McMinn, P.; 2011. "Search-Based Software Testing: Past, Present and Future," In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW '11)*. IEEE Computer Society, Washington, DC, USA, 153-163. DOI=10.1109/ICSTW.2011.100 <http://dx.doi.org/10.1109/ICSTW.2011.100>.
- Metropolis, N.; e Ulam, S.; (1949). "The Monte Carlo Method," *Journal of the American Statistical Association* (American Statistical Association) 44 (247): 335–341. [doi:10.2307/2280232](https://doi.org/10.2307/2280232).
- Metropolis, N.; Rosenbluth, A.W.; Rosenbluth, M.N.; Teller, A.H.; e Teller, E.; (1953), "Equations of State Calculations by Fast Computing Machines," *J. Chem. Phys.*, 21, pp. 1087- 1092.
- Mette, A.; e Hass, J.; "Testing Processes," *icstw*, pp.322-327, 2008 *IEEE International Conference on Software Testing Verification and Validation Workshop*, 2008.
- Michael, A.C.; McGraw, G.; e Schatz, M.A.; 2001. "Generating Software Test Data by Evolution," *IEEE Trans. Softw. Eng.* 27, 12, Dezembro, 2001, p.1085-1110.
- Lyu, M.R.; 2007. "Software Reliability Engineering: A Roadmap," In *2007 Future of Software Engineering (FOSE '07)*. IEEE Computer Society, Washington, DC, USA, 153-170. DOI=10.1109/FOSE.2007.24 <http://dx.doi.org/10.1109/FOSE.2007.24192.384195> <http://doi.acm.org/10.1145/384192.384195>.
- Michalewicz, Z.; 1996. "*Genetic Algorithms + Data Structures = Evolution Programs*," (3rd Ed.). Springer-Verlag, London, UK.
- Michalewicz, Z.; e Fogel, D.B.; (2000), "How To Solve It: Modern Heuristics," *Springer-Verlag*, Berlin.
- Miller, K.W.; Morell, L.J.; Noonan, R.E.; Park, S.K.; Nicol, D.M.; Murrill, B.W.; e Voas, J.M.; 1992. "Estimating the Probability of Failure When Testing Reveals No Failures," *IEEE Trans. Softw. Eng.* 18, 1 (Janeiro 1992), 33-43. DOI=10.1109/32.120314 <http://dx.doi.org/10.1109/32.120314>
- Mills, H.; Dyer, M.; e Linger, R.; "Cleanroom software engineering," *IEEE Software*, pp. 19-25, Setembro. 1987.
- Miller, W.; e Spooner, D.L.; "Automatic Generation of Floating-Point Test Data," *IEEE Transactions on Software Engineering*, Vol. SE-2, No.3, p. 223-226, Setembro, 1976.
- Morell, L.J.; "A Theory of Fault-Based Testing," *IEEE Trans. on Sof. Eng.*, Vol 16, No. 8, Agosto, 1990, pp 844-857.
- Müller, M.; e Pfahl, D.; 2008. "Simulation Methods," In: *Guide to Advanced Empirical Software Engineering*.
- Murugesan, S.; "Dependable software through fault tolerance," *Fourth IEEE Region 10 International Conference*, Novembro, 1989, p.391-399.
- Musa, J.D.; Iannino, A.; e Okumoto, K.; "Software Reliability: Measurement, Prediction, Application," New York: *McGraw-Hill*, 1987.

- Myers, G.J., "The Art of Software Testing," New York, *Addison-Wesley*, 1979.
- Ntafos, S.C.; "On Required Element Testing," *IEEE Trans. Software Eng.*, Vol. SE 10, pp. 795-803, Nov. 1984.
- Ntafos, S.C., "A Comparison of Some Structural Testing Strategies," *IEEE Trans. Software Eng.*, Vol. 14, No. 6, pp. 868-873, Jun. 1988.
- Ntafos, S.C.; "On random and partition testing," *SIGSOFT Softw. Eng. Notes* 23, 2 (Março 1998), 42-48.
- Ntafos, S.C.; "On Comparisons of Random, Partition, and Proportional Partition Testing," *IEEE Trans. Softw. Eng.* 27, 10 (Outubro 2001), 949-960.
- Nie, C.; e Leung, H.; "A survey of combinatorial testing," *ACM Comput. Surv.* 43, 2, Article 11 (Fevereiro 2011), 29 paginas, 2011.
- Offutt, J.; Rothmel, G.; e Zapf, C.; "An Experimental Evaluation of Selective Mutation," in *Proceedings of the 15th International Conference on Software Engineering (ICSE'93)*. Baltimore, Maryland: IEEE Computer Society Press, Maio 1993, pp. 100-107.
- Offutt, J.; e Hayes, J.H.; 1996. "A semantic model of program faults," *SIGSOFT Softw. Eng. Notes* 21, 3 (Maio 1996), 195-200. DOI=10.1145/226295.226317 <http://doi.acm.org/10.1145/226295.226317>.
- Offutt, J.; "An integrated automatic test data generation system," *Journal of Systems Integration*, Volume 1, N 3-4, 391-409, Abril 1991.
- Offutt, J.; Jin, Z.; e Pan, J.; "The dynamic domain reduction procedure for test data generation," *Softw. Pract. Exper.* 29, 2, 167-193. Fevereiro, 1999.
- Offutt, J.; e Abdurazik, A.; "Generating tests from UML specifications," In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 416-429, Fort Collins, CO, Outubro 1999. IEEE Computer Society Press.
- Oh, N.; Mitra, S.; e McCluskey, E.J.; 2002. "ED4I: Error Detection by Diverse Data and Duplicated Instructions," *IEEE Trans. Comput.* 51, 2 (Fevereiro 2002), 180-199. DOI=10.1109/12.980007 <http://dx.doi.org/10.1109/12.980007>.
- Open Source Software Testing Tools (OSSTT), News and Discussion. <http://www.opensourcetesting.org/>. Consultado em Abril de 2011.
- Oprea, M.; e Forrest, S.; (1998), "Simulated Evolution of Antibody Gene Libraries Under Pathogen Selection," In *Proc. of the IEEE SMC'98*.
- Oprea, M.; e Forrest, S.; (1999), "How the Immune System Generates Diversity: Pathogen Space Coverage with Random and Evolved Antibody Libraries," In *Proc. of the GECCO'99*, 2, pp. 1651-1656.
- Ostrand, T.J.; e Balcer, M.J.; "The category-partition method for specifying and generating functional tests," *Commun ACM*, 1988, 31, 6 (Junho), 676-686.
- Ostresh, J.M.; Husar, G.M.; Blondelle, S.E.; Dörner, B.; Weber, P.A.; e Houghten, R.A.; "Libraries from libraries: chemical transformation of combinatorial libraries to extend the range and repertoire

of chemical diversity,” *Proc. Nati. Acad. Sci., USA*, Vol. 91, pp. 11138-11142, Novembro, 1994.Chemistry.

Pacheco, C.;Lahiri, S.K.; Ernst, M.D.; e Ball, T.; “Feedback Directed Random Test Generation,” In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, (Minneapolis, MN, USA), 2007.

Padua Filho, W.P.; “Engenharia de Software,” Ed. *LTC*, 2ª Ed., 2003.

Pande, H.D.; Landi, W.A.; e Ryder, B.; “InterproceduralDef-Use Associations for C Systems with Single Level Pointers,” *IEEE Tras.Soft.Eng.*, Vol. 20, No. 5, pp.385-403, Maio, 1994.

Pargas, R.P.;Harrold, M.J.;e Peck, R.R.;“Test-data generation using genetic algorithms,” *Software Testing Verification and Reliability*,1999, Vol: 9, Issue: 4, Wiley, p.: 263-282.

Pedrycz,W.;e Peters, J.F.; 1998. “*Computational Intelligence in Software Engineering*,” *World Scientific Publishing Co., Inc.*, River Edge, NJ, USA.

Pedrycz, W.; 2002. “Computational intelligence as an emerging paradigm of software engineering,” In *Proceedings of the 14th international conference on Software engineering and knowledge engineering (SEKE '02)*. ACM, New York, NY, USA, 7-14. DOI=10.1145/568760.568763 <http://doi.acm.org/10.1145/568760.568763>.

Perelson,S.;e Oster,G.F.;“Theoretical studies of clonal selection: minimal antibody repertoire size and reliability of self-nonsel self discrimination,” *In J. Theor. Biol.*, Vol. 81 (1979), pp. 645-670.

Peters, K.D.; e Parnas, D.L.; “Using test oracles generated from program documentation,” *IEEE Transactions on Software Engineering* 1998; 24(3):161–173.

Pinto, G.H.L.; e Vergilio, S.R.; “A Multi-Objective Genetic Algorithm to Test Data Generation,”*In: IEEE International Conference on Tools with Artificial Intelligence, ICTAI* 2010.

Poston, R. M.; Sexton, M. P.; “Evaluating and Selecting Testing Tools,” *IEEE Software*,Maio 1992, pp 33-42.

Press, W.H.;Flannery,B.P.;Teukolsky, S.A.;e Vetterling,W.T.;“*Numerical Recipes in C: The Art of Scientific Computing*,” *CambridgeUniversityPress*, Cambridge, UK, second edition, 1992.

Pressman, R.S.; “Software Engineering: A Practitioner’s Approach,” *MacGraw-Hill*, New York, 2006.

Ramamoorthy, C.V.; Siu-Bun, F.H.; e Chen, W.T.; “On Automated Generation of Program Test Data,”*IEEE Transactions on Software Engineering*, Vol. SE-2, N. 4, P. 293-300, Dezembro, 1976.

Randell,A.;1975.“System structure for software fault tolerance,” *In Proceedings of the international conference on Reliable software*. ACM, New York, NY, USA, 437-449. DOI=10.1145/800027.808467 <http://doi.acm.org/10.1145/800027.808467>.

Rapps, S.; e Weyuker, E.J.; “Selecting Software Test Data Using Data Flow Information,” *IEEE Trans. Software Eng.*, Vol. 11, pp. 367-375, Abril 1985.

Rosenberg, J.;(2008),“Statistical Methods and Measurement,”*In: Guide to Advanced Empirical Software Engineering*, 2008, Springer-Verlag New York, Inc., Secaucus, NJ, USA, pp. 155-184.

Rothermel, G.; Untch, R.H.; Chu, C.; Harrold, M.J.; 1999. "Test Case Prioritization: An Empirical Study," In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*. IEEE Computer Society, Washington, DC, USA. 1999.

Ryder, B.G.; "Constructing the Call Graph of a Program," *IEEE Trans. Software Engineering*, Vol SE-5, No. 3, Maio 1979, pp.216-226.

Saglietti, F.; "Software Diversity Metrics: Quantifying Dissimilarity in the Input Partition," *Software Engineering Journal*, Janeiro 1990, pp. 59 -63.

Santiago Júnior, V.A.; Cristiá, M.; Vijaykumar, N.L.; "Model-Based Test Case Generation Using Statecharts And Z: A Comparison And A Combined Approach," *Relatório Técnico INPE-16677-RPQ/850*, (<http://mtc-m19.sid.inpe.br/col/sid.inpe.br/mtc-m19%4080/2010/02.26.14.05/doc/publicacao1.pdf>), 2010.

Santini, S.; e Jain, R.; 1999. "Similarity Measures," *IEEE Trans. Pattern Anal. Mach. Intell.* 21, 9 (Setembro 1999), 871-883. DOI=10.1109/34.790428 <http://dx.doi.org/10.1109/34.790428>

Sarma, M.; Murthy, P.V.R.;Jell, S.; e Ulrich, A.; "Model-based testing in industry: a case study with two MBT tools," In *Proceedings of the 5th Workshop on Automation of Software Test (AST '10)*. ACM, New York, NY, USA, 87-90, 2010.

Schneckenburger, C.; e Mayer, J.; 2007. "Towards the determination of typical failure patterns," In *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting (SOQUA '07)*. ACM, New York, NY, USA, 90-93. DOI=10.1145/1295074.1295091 <http://doi.acm.org/10.1145/1295074.1295091>

SEI, CMMI for Development, Version 1.2, Carnegie Mellon University, *Software Engineering Institute*, CMU/SEI 2006-TR-008 ESC-TR-2006-008, Agosto, 2006.

Shafique, M.; e Labiche, Y.; "A Systematic Review of Model Based Testing Tool Support," Technical Report SCE-10-04, *Carleton University*, Maio 2010.

Shanon, C.; 1948. "A mathematical theory of communication," *Bell System Tech. J.* 27: 379_423, 623_656.

Shull, F.; Singer, J.; e Sjøberg, D.I.K.; 2008. "*Guide to Advanced Empirical Software Engineering*," *Springer-Verlag* New York, Inc., Secaucus, NJ, USA.

Simão, A.S.; "Teste Baseado em Modelos," in: Delamaro M.E., Maldonado J.C., Jino M., (Eds.) "Introdução ao Teste de Software," *Elsevier Editora Ltda.*, 2007.

Sommerville, I.; "Software Engineering," 7th ed., *Addison Wesley*; Maio 2006.

Srinivas, M.; e Patnaik, L.M.; 1994. "Genetic algorithms: A Survey," *Computer* 27, 6 (Junho 1994), 17-26. <http://dx.doi.org/10.1109/2.294849>.

Stahl, T.; e Völter, M.; "Model-Driven Software Development: Technology, Engineering, Management," *Wiley*, 2006.

Strehl, A.; Ghosh, J.; e Mooney, R.J.; "Impact of similarity measures on web-page clustering," In *Proc. AAAI Workshop on AI for Web Search (AAAI 2000)*, Austin, p. 58-64. AAAI/MIT Press, Julho 2000.

SWEBOK "Guide to the Software Engineering Body of Knowledge," *IEEE Computer Society*, 2004.

Tai, K-C.; "Theory of Fault-Based Predicate Testing for Computer Programs," *IEEE Trans. Softw. Eng.* 22, 8 (Agosto 1996), 552-562. 1996.

Thayer, R.; Lipow, M.; e Nelson, E.; "Software Reliability," *Amsterdam: North-Holland*, 1978.

Thompson, M.C.; Richardson, J.R.; e Clarke, L.A.; "An information flow model of fault detection," *SIGSOFT Softw. Eng. Notes* 18, 3 (Julho 1993), 182-192.

Tonella, P.; "Evolutionary testing of classes," *In Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. Vol. SE-8(4), p. 119-28, ACM Press, Boston, Massachusetts, USA, 2004.

Tracey, N.; Clark, J.; e Mander, K.; "Automated program flaw finding using simulated annealing," *SIGSOFT Softw. Eng. Notes* 23, 2 (Março 1998), 73-81, 1998.

Tracey, N.; Clark, J.; Mander, K.; e McDermid, J.; "Automated test-data generation for exception conditions," *Softw. Pract. Exper.* 30, 1, Janeiro, 2000, 61-79.

Trojanowski, K.; e Michalewicz, Z.; "Evolutionary Algorithms and the Problem-Specific Knowledge," *Proceedings of the 2nd National Conference on Evolutionary Computation and Global Optimisation*, Rytro, Poland, Warsaw Univ. of Technology Press, pp. 281-292 (1997).

Utting, M.; e Legeard, B.; "Practical Model-Based Testing: A Tools Approach," *Morgan Kaufmann Publishers Inc.*, San Francisco, CA, USA, 2006.

Utting, M.; Pretschner, A.; e Legeard, B.; "A Taxonomy of Model-Based Testing," Working Paper, Dep. of Computer Science, *University of Waikato*, 2006.

Vergilio, S.R.; "Caminhos Não Executáveis: Caracterização, Previsão e Determinação para Suporte ao Teste de Programas," Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas, SP. Janeiro 1992.

Vergilio, S.R.; Maldonado, J.C.; e Jino, M.; "Caminhos Não Executáveis na Automação das Atividades de Teste," *VI Simpósio Brasileiro de Engenharia de Software*, p. 343-456, Gramado, RS, Novembro, 1992.

Vergilio, S.R.; "Critérios Restritos de Teste de Software: Uma Contribuição para Gerar Dados de Teste mais Eficazes," Tese de Doutorado, DCA/FEE/UNICAMP, Campinas, SP 1997.

Vergilio, S.R.; Colanzi, T.E.; Pozo, A.T.R.; e Assuncao, W.K.G.; 2011. "Search Based Software Engineering: A Review from the Brazilian Symposium on Software Engineering," *In Proceedings of the 2011 25th Brazilian Symposium on Software Engineering (SBES '11)*. IEEE Computer Society, Washington, DC, USA, 50-55. DOI=10.1109/SBES.2011.13 <http://dx.doi.org/10.1109/SBES.2011.13>

Vergilio, S.R.; Bueno, P.M.S.; Dias Neto, A.C.; Martins, E.; Vincenzi, A.; "Tutorial: Geração de Dados de Teste: Principais Conceitos e Técnicas," *In: Anais do II Congresso Brasileiro de Software: Teoria e Prática*, 2011, São Paulo.

Vilela, P.R.S.; "Critérios potenciais usos de integração: definição e análise," Tese de Doutorado, DCA/FEE/UNICAMP, Campinas, SP, Abril 1998.

- Voas, J.M.; “PIE: A Dynamic Failure-Based Technique,” *IEEE Trans. on Software Eng.*, Agosto, 1992, pp. 717-727.
- Von Zuben, F.J.; (2000), “Computação Evolutiva: Uma Abordagem Pragmática,” *DCA/FEEC, Unicamp* (at. ftp://ftp.dca.fee.unicamp.br/pub/docs/vonzuben/ia013_1s07/tutorialEC.pdf).
- Von Zuben, F.J.; e De Castro, L.N.; “Tópico 1 – Hill Climbing e Simulated Annealing,” material de aula IA707- Computação Evolutiva, *DCA/ FEEC, Unicamp*, (versão de 2002).
- Walton, G.H.; e Poore, J.H.; “Generating transition probabilities to support model-based software testing,” *Software: Practice and Experience*, 30(10):1095-1106, Agosto 2000.
- Wegener, J.; Sthamer, H.; Jones, B.F.; e Eyres, D.E.; “Testing real-time systems using genetic algorithms,” *Software Quality Journal*, Volume 6, Número 2 (1997), 127-135, DOI:10.1023/A:1018551716639.
- Wegener, J.; Buhr, K.; e Pohlheim, H.; “Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing,” *In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '02)*, Morgan Kaufmann Publishers Inc., USA, 2002, p. 1233-1240.
- Weyuker, E.L.; Goradia, T.; e Singh, A.; 1994. “Automatically Generating Test Data from a Boolean Specification,” *IEEE Trans. Softw. Eng.* 20, 5, P. 353-363, Maio, 1994.
- Weyuker, E.L.; “The Cost of Data Flow Testing: An Empirical Study,” *IEEE Trans. Software Eng.*, Vol. 16, No. 2, pp. 121-128, Fevereiro 1990.
- Weyuker, E.L.; “More Experience with Data Flow Testing,” *IEEE Trans. Software Eng.*, Vol. 19, No. 9, pp. 912-919, Setembro 1993.
- Weyuker, E.J.; “On testing non-testable programs,” *The Computer Journal*, 1982; 25(4):465–470.
- Weyuker, E.J.; e Jeng, B.; “Analyzing partition testing strategies,” *IEEE Trans. Softw. Eng.* 17, 7 (Julho), 703–711, 1991.
- White, L.J.; Cohen, E.I.; “A Domain Strategy for Computer Program Testing,” *IEEE Trans. on Soft. Eng.*, Vol SE-6, No.3, Maio 1980, pp. 247-257.
- Whittaker, J.A.; “What Is Software Testing? And Why Is It So Hard?,” *IEEE Software*. 17, 1, Janeiro, 2000, 70-79.
- Whittaker, J.A.; e Voas, J.; 2000. “Toward a More Reliable Theory of Software Reliability,” *Computer* 33, 12 (Dezembro 2000), 36-42. DOI=10.1109/2.889091 <http://dx.doi.org/10.1109/2.889091>.
- Wichmann, B.A.; “Some Remarks About Random Testing,” em: http://www.npl.co.uk/scientific_software/publications/validation/random_testing.pdf, consultado em Abril, 2011. *National Physical Laboratory*.
- Willett, P.; “Chemoinformatics – similarity and diversity in chemical libraries,” *Current Opinion in Biotechnology*, Vol 1, Issue 1, Fevereiro, 2000, PP. 85-88.
- Williams, A.W.; e Probert, R.L.; “Formulation of the interaction test coverage problem as an integer program,” *In Proceedings of the IFIP 14th International Conference on Testing Communicating Systems, XIV (TestCom'02)*. Kluwer, 283. 2002.

Windisch, A.; Wappler, S.; e Wegener, J.; “Applying particle swarm optimization to software testing,” *In Proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO '07)*. ACM, New York, NY, USA, 2007, p.1121-1128.

Witkin, A.; “An introduction to physically based modeling: particle system dynamics,” Technical Report, *Robotics Institute, Carnegie Mellon University*, Agosto, 1997, (at: <http://www.cs.cmu.edu/~baraff/pbm/particles.pdf>).

Wong, W.E.; Horgan, J.R.; London, S.; e Mathur, A.P.; “Effect of test set size and block coverage on fault detection effectiveness,” *in Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering*, pp230-238, Monterey, CA, Novembro 1994.

Yang, Q.; Li, J.; e Weiss, D.; “A survey of coverage based testing tools,” *In Proceedings of the 2006 international workshop on Automation of software test (AST '06)*. ACM, New York, NY, USA, 99-103. 2006.

Yano, T.; Martins, E.; e De Sousa, F.L.; “Generating Feasible Test Paths from an Executable Model Using a Multi-objective Approach,” *In Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE Computer Society, Washington, DC, USA, 236-239. 2010.

Zeil, S.J.; Afifi, F.H.; e White, L.J.; 1992. “Detection of linear errors via domain testing,” *ACM Trans. Softw. Eng. Methodol.* 1, 4 (Outubro 1992), 422-451. DOI=10.1145/136586.136590 <http://doi.acm.org/10.1145/136586.136590>.

Zhan, Y.; e Clark, J.A.; “Search-based mutation testing for Simulink models,” *In Proceedings of the 2005 conference on Genetic and evolutionary computation (GECCO '05)*, ACM, New York, NY, USA, 1061-1068, 2005.

Zhao, R.; Harman, M.; e Li, Z.; “Empirical Study on the Efficiency of Search Based Test Generation for EFSM Models,” *In Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW '10)*, IEEE Computer Society, Washington, DC, USA, 222-231, 2010.

Zhu, H.; e Hall, P.A.V.; “Test Data Adequacy Measurement,” *Software Eng. Journal*, Jan. 1993, pp. 21-29.

Zitzler, E.; Laumanns, M.; e Bleule, S.; “A Tutorial on Evolutionary Multiobjective Optimization,” (2003), *In Metaheuristics for Multiobjective Optimisation*, p. 3-38, Springer.