Avaliação da efetividade dos critérios de teste estruturais no contexto de programas concorrentes

Maria Adelina Silva Brito

Avaliação da efetividade dos critérios de teste estruturais no contexto de programas concorrentes

Maria Adelina Silva Brito

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

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

USP – São Carlos
Dezembro de 2011

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

A862a Adelina Silva Brito, Maria
Avaliação da efetividade dos critérios de teste estruturais no contexto de programas concorrentes / Maria Adelina Silva Brito; orientadora Simone do Rocio Senger de Souza. -- São Carlos, 2011.
94 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2011.

1. Teste de Software. 2. Programas Concorrentes. 3. MPI. 4. Engenharia de Software Experimental. I. do Rocio Senger de Souza, Simone, orient. II. Título.

Agradecimentos

Agradeço a Deus por todos os momentos de esperança e fé, por proteger e guiar minha vida até aqui pelos caminhos mais seguros.

A toda minha família pelo amor, apoio e auxílio na vinda para São Carlos. Em especial meus pais Fidelino e Maria do Alívio, meus irmãos Vagno, Flávia e Wailson, e meu namorado Erinaldo, sem vocês o mestrado não seria possível.

À minha orientadora Simone do Rocio Senger de Souza e ao professor Paulo Sérgio, pela oportunidade, profissionalismo, incentivo, orientação e paciência, fatores que muito contribuíram para a conclusão deste trabalho.

Muito obrigada aos meus amigos do LabES e agregados: Lucas, Rafael, Adriano, David, Endo, Paulo Nardi, Marcão, Faimison, André, Rafael, Neiza, Draylson, Joyce, Vânia, Katia, Silvana, Vinicius, Bruno Cafeo, e os demais. Ao colega da estatística, João, sempre atencioso com os cálculos estatísticos. Obrigada aos Professores e funcionários. Aos amigos da Bahia, em especial Kamila pelos momentos de alegria e por tornar a estadia em São Carlos mais agradável. Obrigada de coração!

À FAPESP pelo apoio financeiro que viabilizou o trabalho.

Resumo

A Engenharia de Software tem desenvolvido técnicas e métodos para apoiar o desenvolvimento de software confiável, flexível, com baixo custo de desenvolvimento e fácil manutenção. Técnicas e critérios de teste contribuem nessa direção, fornecendo mecanismos para produzir software com alta qualidade. Este trabalho apresenta um estudo experimental para avaliar o custo, eficácia e a dificuldade de satisfação (*strength*) dos critérios estruturais propostos para programas concorrentes. Esta avaliação foi conduzida usando oito programas implementados em MPI e utilizando a ferramenta de teste ValiMPI. Com base em taxonomias, defeitos foram injetados nos programas de modo a avaliar a eficácia dos critérios de teste em revelar os defeitos inseridos. Os resultados obtidos demonstraram o aspecto complementar dos critérios e informações sobre o custo e eficácia, que contribuíram para o estabelecimento de uma estratégia de teste incremental para aplicar os critérios de teste em uma boa relação custo-eficácia. Para concluir, os resultados indicam que os critérios de teste estrutural propostos para programas concorrentes em MPI são promissores e podem auxiliar a detectar defeitos nessas aplicações, melhorando a qualidade das mesmas.

Abstract

The software engineering has developed techniques and methods to help the software development reliable, flexible and with lower development costs and easy maintenance. Testing techniques and criteria contribute in this sense, providing mechanisms to produce high quality software. This work presents an experimental study to evaluate the cost, effectiveness and strength of the structural testing criteria for concurrent programs. This evaluation has been conducted using eight programs implemented in MPI and the ValiMPI testing tool. Based on a fault taxonomy for concurrent programs, defects have been injected in these programs in order to evaluate the effectiveness of the testing criteria. The results indicate the complementary aspect of the criteria and the information about cost and effectiveness have contributed to the establishment of an incremental testing strategy to apply the testing criteria in good relation cost-effectiveness. To conclude, the results indicate that the structural testing criteria, proposed for MPI concurrent programs, are promising and they can help to find defects in these applications, improving their quality.

Sumário

1 Introdução **1**
1.1 Contextualização . . . 1
1.2 Motivação . . . 5
1.3 Objetivo . . . 6
1.4 Organização . . . 6

2 Fundamentação Teórica **7**
2.1 Considerações Iniciais . . . 7
2.2 Teste de Software . . . 7
2.2.1 Técnica de Teste Funcional . . . 9
2.2.2 Técnica de Teste Estrutural . . . 9
2.2.3 Técnica de Teste Baseado em Erros . . . 13
2.3 Programas Concorrentes . . . 14
2.3.1 O Padrão MPI . . . 16
2.4 Engenharia de Software Experimental . . . 18
2.4.1 O Processo Experimental . . . 19
2.4.2 Experimentos em Teste de Software . . . 20
2.5 Considerações Finais . . . 22

3 Teste de Programas Concorrentes **23**
3.1 Considerações Iniciais . . . 23
3.2 Teste Estrutural de Programas Concorrentes . . . 23
3.2.1 A Ferramenta ValiMPI . . . 27
3.3 Trabalhos Relacionados ao Teste de Programas Concorrentes . . . 34
3.4 Considerações Finais . . . 37

4 Definição e Planejamento do Experimento Realizado **39**
4.1 Considerações Iniciais . . . 39
4.2 Definição do Experimento . . . 39
4.2.1 Objetivo . . . 40
4.2.2 Metas . . . 40
4.3 Planejamento . . . 40
4.3.1 Seleção do Contexto . . . 40

4.3.2 Hipóteses . . . 43
4.3.3 Seleção dos Indivíduos . . . 44
4.3.4 Variáveis . . . 44
4.3.5 Descrição da Instrumentação do Experimento . . . 45
4.3.6 Validade . . . 46
4.4 Considerações Finais . . . 47

5 Preparação e Execução do Experimento Realizado 49
5.1 Considerações Iniciais . . . 49
5.2 Preparação do Experimento . . . 49
5.2.1 Ambiente de Testes . . . 50
5.2.2 Descrição dos Programas . . . 50
5.3 Execução do Experimento . . . 51
5.3.1 Geração dos Conjuntos de Casos de Teste . . . 51
5.3.2 Semeadura de Defeitos . . . 51
5.3.3 Execução dos Programas . . . 52
5.4 Considerações Finais . . . 53

6 Análise dos Resultados 57
6.1 Considerações Iniciais . . . 57
6.2 Estatística Descritiva . . . 57
6.2.1 Análise do Custo dos Critérios . . . 58
6.2.2 Análise da Eficácia . . . 62
6.2.3 Análise do *Strength* . . . 64
6.3 Redução do Conjunto de Dados . . . 70
6.4 Teste de Hipóteses . . . 71
6.4.1 Análise do Custo . . . 71
6.4.2 Análise da Eficácia . . . 76
6.4.3 Análise do *Strength* . . . 78
6.5 Síntese das Análises . . . 80
6.6 Considerações Finais . . . 82

7 Conclusão 83
7.1 Caracterização da Pesquisa Realizada . . . 83
7.2 Contribuições . . . 84
7.3 Dificuldades e Limitações . . . 84
7.4 Trabalhos Futuros . . . 84

Referências 94

Lista de Figuras

2.1 Código do programa Identifier 11
2.2 GFC do programa Identifier 12
2.3 Código de um programa exemplo ilustrando o não determinismo 17
2.4 O processo de experimentação e suas fases (Wohlin et al., 2000) 19

3.1 Arquitetura da ValiMPI 28
3.2 Código do programa GCD 31
3.3 GFCP do programa GCD 32
3.4 Elementos requeridos para o critério todas-arestas-s 33
3.5 Saída do ValiEval para a cobertura do critério todas-arestas-s 34

6.1 Custo para cada critério considerando os programas 58
6.2 Custo para cada programa considerando os elementos requeridos 59
6.3 Custo para cada critério considerando os elementos não executáveis 61
6.4 Custo para cada critério considerando os casos de teste adequados 62
6.5 Boxplot para a eficácia dos critérios 63
6.6 Eficácia para os critérios de teste 64
6.7 *Strength* para o conjunto de teste adequado ao critério todos-nos 65
6.8 *Strength* para o conjunto de teste adequado ao critério todos-nos-r 66
6.9 *Strength* para o conjunto de teste adequado ao critério todos-nos-s 67
6.10 *Strength* para o conjunto de teste adequado ao critério todas-arestas 68
6.11 *Strength* para o conjunto de teste adequado ao critério todas-arestas-s . . . 69
6.12 *Strength* para o conjunto de teste adequado ao critério todos-c-usos 70
6.13 *Strength* para o conjunto de teste adequado ao critério todos-p-usos 71
6.14 *Strength* para o conjunto de teste adequado ao critério todos-s-usos 72
6.15 Resultados do teste de Tukey para o custo dos critérios com base no nº de elementos requeridos 73
6.16 Boxplot para o custo dos critérios com base no número de elementos requeridos 74
6.17 Resultados do teste de Tukey para o custo com base no nº de elementos requeridos não executáveis 75
6.18 Resultados do teste Kruskal-Wallis para o custo com base no nº de casos de teste adequados 76

6.19 Resultados do teste de Kruskal-Wallis para a eficácia dos critérios 77
6.20 Dendrograma para o *strength* do conjunto todos-nos 78
6.21 Dendrograma para o *strength* do conjunto todos-nos-r 79
6.22 Dendrograma para o *strength* do conjunto todas-arestas 79
6.23 Dendrograma para o *strength* do conjunto todos-c-usos 80
6.24 Dendrograma para o *strength* do conjunto todos-p-usos 81
6.25 Dendrograma para o *strength* do conjunto todos-s-usos 81

Lista de Tabelas

4.1 Defeitos inseridos nos programas do experimento 42

5.1 Características dos programas utilizados no experimento 50
5.2 Elementos requeridos e não executáveis 51
5.3 Tamanho dos conjuntos adequados 51
5.4 Total de defeitos inseridos em cada programa 52
5.5 Dados sobre a eficácia de cada critério de teste 53
5.6 *Strength* para o programa GCD 53
5.7 *Strength* para o programa Jacobi 53
5.8 *Strength* para o programa Mmult 54
5.9 *Strength* para o programa Qsort 54
5.10 *Strength* para o programa Reduction 54
5.11 *Strength* para o programa Sieve 54
5.12 *Strength* para o programa Trap 54
5.13 *Strength* para o programa Van der Waals 55

6.1 Análise do custo considerando os elementos requeridos para cada critério . 60
6.2 Resultados da ANOVA para o custo com base no número de elementos requeridos para cada critério 70
6.3 Resultados da ANOVA para o custo com base no número de elementos não executáveis 74

Capítulo

1

Introdução

1.1 Contextualização

Teste de software é uma das atividades de VV&T (Verificação, Validação e Teste de Software), utilizada para aumentar a garantia de qualidade do software que está sendo desenvolvido. O teste é uma atividade dinâmica, cujo objetivo é revelar erros existentes no software em teste, caso existam. Essa atividade deve ser realizada durante o ciclo de desenvolvimento, sendo composta por fases como: planejamento, projeto dos casos de teste, execução dos testes e análise dos resultados.

A qualidade dos testes é determinada pelos casos de teste utilizados. Em geral, o programa em teste deveria ser submetido a todos os valores do domínio de entrada, porém, a realização de testes dessa forma é inviável em razão do grande número de valores de entrada possíveis. Em decorrência dessa limitação foram criados os critérios de teste, os quais contribuem com a seleção sistemática de casos de teste garantindo que partes específicas do programa sejam testadas.

A diferença entre os critérios de teste é a origem da informação utilizada para estabelecer os subdomínios e construir os conjuntos de teste. Os critérios de teste encontram-se agrupados nas técnicas: funcional, estrutural e baseada em erros. Para o teste funcional, utiliza-se para geração dos casos de teste a especificação do programa, para o teste estrutural, utiliza-se o código fonte e para o teste baseado em erros são utilizados os erros mais comuns cometidos durante o processo de desenvolvimento.

Nos programas tradicionais (programas com características sequenciais), estudos realizados ao longo dos anos possibilitaram a criação de técnicas e critérios para atividades de validação, os quais consideram duas questões importantes: a seleção de casos de teste e a avaliação da adequação dos casos de teste em relação ao programa que está sendo testado (Coward, 1988; Howden, 1975; Laski e Korel, 1983; Maldonado, 1991; Myers et al., 2004; Rapps e Weyuker, 1985; Ural e Yang, 1988). Diferentemente dos programas tradicionais, os programas concorrentes possuem características como comunicação, sincronização, concorrência e não determinismo que tornam a atividade de teste mais complexa.

A programação concorrente é considerada essencial para redução do tempo computacional em vários domínios, como: previsão de tempo, dinâmica de fluídos, processamento de imagens, química quântica e outros. Como a demanda por esse tipo de software é crescente, cresce também o número de aplicações para os ambientes multicomputadores e multiprocessadores. As aplicações concorrentes são específicas a esse tipo de sistemas e são formadas por um conjunto de processos que se comunicam para solução de um problema. A construção de processos concorrentes requer o uso das seguintes primitivas para permitir a comunicação e sincronização entre os processos (Almasi e Gottlieb, 1994): 1) primitivas para definir quais processos executarão em paralelo; 2) iniciar e finalizar processos concorrentes e; 3) coordenar e especificar a interação entre os processos concorrentes, enquanto estes estiverem em execução. Dessas primitivas, destacam-se as necessárias à interação (comunicação e sincronização) entre os processos, em razão da frequência de utilização e impacto no desempenho final do programa concorrente.

Almasi e Gottlieb (1994) citam três formas de implementar programas concorrentes: 1) geração de código concorrente a partir de código sequencial, como algumas versões de compiladores para *Fortran* e *C*; 2) uso de linguagens de programação concorrente, como *Unified Parallel C* e *High Performance Fortran*; e 3) extensões para linguagens sequenciais, como *C* e *Fortran*, implementadas por meio de ambientes de passagem de mensagens. As extensões usam bibliotecas para permitir comunicação entre processos concorrentes. O PVM - *Parallel Virtual Machine* (Beguelin, 1993) e o MPI - *Message Passing Interface* (Snir et al., 1996) são exemplos de ambientes de passagem de mensagem.

A interação entre processos ocorre segundo dois paradigmas distintos, normalmente, vinculados a organização de memória disponível (Almasi e Gottlieb, 1994; Hwang e Briggs, 1984). No paradigma de memória compartilhada os processos concorrentes acessam o mesmo espaço de endereçamento e assim, compartilham variáveis de memória. Nesse modelo, a sincronização pode ocorrer explicitamente por meio de construções como semáforos ou monitores (Tanenbaum, 2001). No paradigma de passagem de mensagens, é usada memória distribuída e os processos concorrentes acessam apenas seu espaço de endereçamento, ou seja, não compartilham variáveis. A sincronização com memória distribuída

ocorre por meio de troca de mensagens, as quais podem ser representadas usando primitivas *send/receive* ou encapsuladas em estruturas como RPC (*Remote Procedure Calls*), RMI (*Remote Method Invocation*), entre outras. Vale ressaltar que a execução de aplicações concorrentes pode utilizar os paradigmas de passagem de mensagens e memória compartilhada na mesma aplicação. Isso pode acontecer com *threads* iniciadas no mesmo processador ou com o uso de plataformas heterogêneas que incluam memória compartilhada e distribuída.

Diferentes iniciativas para validação de programas concorrentes são encontradas, atuando na definição de critérios e ferramentas de teste para memória compartilhada (Chung et al., 1996; Joshi et al., 2009; Stoller, 2002; Yang e Pollock, 2003; Yang et al., 1998; Yang e Chung, 1992), outras relacionam-se a aplicação de critérios ou ferramentas de apoio ao teste de software para o paradigma de memória distribuída (Liang et al., 2000; Souza et al., 2005; Vergilio et al., 2005). Outros trabalhos têm identificado o problema de detecção de condições de disputa (Edelstein et al., 2003; Lei e Carver, 2004) e outros apresentam mecanismos que de alguma forma tratam o não determinismo (Damodaran-Kamal e Francioni, 1993; Harvey e Strooper, 2001; Li et al., 2004; Seo et al., 2001; Wildman et al., 2005; Xufang Gong, 2007). Para grande parte dos trabalhos nota-se ausência de estudos experimentais com o intuito de prover comparações da abordagem proposta com outras existentes. Os trabalhos em sua maioria realizam estudos de casos ou experimentos apenas com o objetivo de validar a abordagem proposta.

A Experimentação em Engenharia de Software tem como objetivo caracterizar, avaliar, prever, controlar ou melhorar tanto os produtos, como também os processos, recursos, modelos ou teorias. Wohlin et al. (2000) afirmam que a experimentação pode proporcionar uma base de conhecimento para reduzir incertezas sobre quais teorias, ferramentas e metodologias são adequadas, como também descobrir novas áreas de pesquisa ou conduzir as teorias para direções promissoras. Quanto mais um experimento for replicado em ambientes e culturas diferentes, mais dados são obtidos sobre o objeto de estudo, o que faz com que sua caracterização seja mais significante.

O estudo sobre técnicas, critérios e ferramentas de teste disponíveis para programas sequenciais, tem se intensificado nos últimos anos, buscando avaliar as diferentes propostas em termos de custo e eficácia em revelar defeitos (Souza et al., 2007a). Essas comparações são fundamentais para o estabelecimento de estratégias de aplicação de modo a aproveitar as vantagens de cada técnica e critério de teste. Nos estudos experimentais, os seguintes fatores são utilizados pra avaliar os critérios de teste: custo, eficácia e dificuldade de satisfação (*strength*) (Souza et al., 2007a). O custo pode ser medido considerando o esforço necessário para que o critério seja usado. O esforço pode ser medido pelo número de casos de teste necessários para satisfazer o critério ou por outras métricas dependentes

do critério, como exemplo, o tempo necessário para identificar caminhos e associações não executáveis, construir manualmente os casos de teste e aprender a utilizar as ferramentas de teste. Eficácia refere-se à capacidade que um critério possui em detectar um maior número de erros em relação a outro. Dificuldade de satisfação ou *strength* refere-se à probabilidade de satisfazer-se um critério tendo sido satisfeito outro critério (Wong, 1993).

Para programas sequenciais alguns experimentos foram conduzidos para avaliação de critérios de teste estruturais, como o estudo conduzido para avaliação dos critérios de fluxo de dados e Potenciais-Usos (Maldonado et al., 1992; Vergilio et al., 1992), obtendo indícios de que na prática esses critérios são factíveis e demandam um número de casos de teste relativamente pequeno. Em Mathur e Wong (1994) compara-se a adequação de conjunto de casos de teste em relação aos critérios Análise de Mutantes e Todos-Usos. O experimento avaliou o *strength* entre os dois critérios, bem como seus custos, já que esses critérios são incomparáveis do ponto de vista teórico. Com o estudo foram obtidos indícios de que na prática o critério Análise de Mutantes inclui o critério Todos-Usos. Outros estudos também compararam os critérios do teste de Mutação com o critério Todos-Usos (Offutt et al., 1996b; Wong et al., 1995), obtendo resultados semelhantes ao de Mathur e Wong (1994). Experimentos comparando o custo do critério Análise de Mutantes com os critérios Potenciais-Usos (Souza et al., 1997), forneceram indícios de que o custo do critério Análise de Mutantes, de acordo com o número de casos de teste necessários para satisfação do critério, foi maior do que o custo dos critérios Potenciais-Usos. Em relação ao *strength*, observou-se que o critério Análise de Mutantes e Todos-Potenciais-Usos são incomparáveis mesmo do ponto de vista experimental. Porém, os critérios Todos-Potenciais-Usos/Du e Todos-Potenciais-du-Caminhos apresentaram maior *strength* que o critério Todos-Potenciais-Usos em relação ao Análise de Mutantes, motivando investigar o aspecto complementar desses critérios quanto à eficácia. Em Malevris e Yates (2006) é estudado o efeito da aplicação de um critério estrutural sobre outro. Em Frankl e Weiss (1993) foi conduzido um experimento para comparar os critérios todos-usos e todas-arestas. Os resultados permitiram verificar que o critério todos-usos foi significativamente mais efetivo do que o critério todas-arestas.

No contexto de programas concorrentes, um estudo experimental descrito em Saini (2005) foi conduzido com o objetivo de analisar os critérios de teste estrutural *All synchronization pair*, *All-branches*, *All decision/condition*, *All du-pair*, *All synchronization-pair and all branches*, *All synchronization-pair and decision/condition*, *All synchronization-pair and all du-pair*, estabelecendo comparações entre eles. O critério Análise de Mutação foi usado para inserir defeitos e possibilitar a análise de eficácia dos critérios. Ao final do experimento as seguintes conclusões foram descritas: o critério *All synchronization pair* foi o menos custoso, porém não foi o mais eficaz. O critério *All branches* foi o mais efi-

caz entre todos os pares de sincronização, porém, com um custo maior. O critério *All decision/condition* foi avaliado como mediano entre o menos custoso e mais robusto. O critério *All synchronization pair and all du-pair* foi o mais custoso entre os critérios e o mais eficaz. Numa análise mais geral, considerou-se que o critério *all sync-pair and all branches* foi o melhor em eficácia e custo.

Assim, no contexto de programas concorrentes estudos experimentais com o objetivo de avalição de critérios de teste são bastante incipientes. Em sua maioria, são utilizadas aplicações clássicas na área para validação de uma proposta, mas são reduzidos os trabalhos com informações que possam repetir experimentos em outros contextos ou que permitam avaliar novas propostas. Esse é um cenário que se pretende investigar nesta proposta de mestrado, caracterizando um estudo experimental para avaliar critérios de teste definidos para programas concorrentes que utilizam memória distribuída.

1.2 Motivação

A cada dia cresce a demanda por software concorrente em razão do grande volume de dados que necessitam de processamento rápido e ao avanço da tecnologia de hardware que provê máquinas cada vez mais eficientes. Testar programas concorrentes de forma eficaz tem se tornado um importante fator para o sucesso dos programas desenvolvidos nesse domínio. Programas concorrentes não podem ser testados como programas tradicionais, pois estes apresentam particularidades restritas a esse tipo de programa, como a comunicação, sincronização, paralelismo e o não determinismo. Essas características incentivam estudos mais detalhados para realização de testes específicos a esse tipo de programa.

Para apoiar o desenvolvimento de software e o ensino de técnicas de teste, é necessário o desenvolvimento de ferramentas que apoiem o teste estrutural de programas concorrentes. O teste de programas concorrentes para memória distribuída é uma área pouco explorada, visto que foram encontrados um número reduzido de ferramentas de apoio ao teste nesse contexto (Lourenço et al., 1997; Sant'Ana, 2001). Além da ferramenta de teste ValiMPI (Souza et al., 2008b) não foram encontrados relatos sobre outra ferramenta de apoio ao teste de programas concorrentes que usam MPI para passagem de mensagem. Além disso, são poucos os estudos experimentais com o objetivo de validação de critérios de teste nesse contexto. Isso motiva pesquisar detalhadamente os trabalhos existentes e investigar a realização de estudos experimentais com o objetivo de contribuir com essa área de pesquisa. A ferramenta de teste ValiMPI auxilia a aplicação de critérios de teste estruturais para programas concorrentes em MPI, no entanto, os critérios de teste atualmente implementados nessa ferramenta não foram completamente avaliados, fazendo-se necessário analisar o custo, a eficácia e o aspecto complementar desses critérios.

1.3 Objetivo

Desenvolvimento de um estudo experimental para avaliar a eficácia, o custo de aplicação e o aspecto complementar dos critérios de teste estruturais definidos para programas concorrentes que utilizam memória distribuída e o padrão de passagem de mensagem MPI.

1.4 Organização

No Capítulo 2 são apresentados conceitos sobre teste de software e Engenharia de Software Experimental. No Capítulo 3 são apresentados fundamentos do teste de programas concorrentes, considerando modelos de programas, critérios e ferramentas de teste. No Capítulo 4 são apresentadas as fases iniciais do estudo experimental conduzido: a definição e o planejamento. No Capítulo 5 são apresentadas as fases de preparação e execução do experimento realizado. No Capítulo 6 são apresentadas a análise e interpretação dos dados obtidos. Por fim, no Capítulo 7 são apresentadas as conclusões do trabalho.

Capítulo

2

Fundamentação Teórica

2.1 Considerações Iniciais

Neste capítulo são apresentados conceitos relacionados à fundamentação teórica do trabalho, os quais são: Teste de Software, Programas Concorrentes e Engenharia de Software Experimental. O conhecimento desses fundamentos faz-se necessário para a compreensão dos aspectos exploradas nos capítulos seguintes.

O capítulo está organizado da seguinte forma: na Seção 2.2 são apresentados os principais conceitos sobre teste de software, considerando as técnicas de teste e características de cada uma, com destaque aos critérios da técnica estrutural. Na Seção 2.3 são apresentados fundamentos sobre teste de programas concorrentes. Na Seção 2.4 é abordada a Engenharia de Software Experimental, e na Seção 2.5 estão as considerações finais deste capítulo.

2.2 Teste de Software

O padrão IEEE. (1990) apresenta algumas definições aplicadas no contexto de teste de software. São elas: defeito (*fault*) é um passo incorreto, processo ou definição de dados em um programa; engano (*mistake*) é uma ação humana capaz de produzir um resultado incorreto; erro (*error*) caracteriza-se por um resultado incorreto e; falha (*failure*) é a inabilidade do sistema executar as funções requeridas com o desempenho necessário.

Atividades de garantia de qualidade de software têm sido inseridas no decorrer do processo de desenvolvimento, como as atividades de VV&T (Verificação, Validação e Teste de Software), com o objetivo de reduzir a ocorrência de defeitos no produto. Essas atividades podem ser estáticas ou dinâmicas. As estáticas são desenvolvidas sem que haja execução de um programa ou modelo, já as dinâmicas exigem a execução de um programa ou de um modelo (Delamaro et al., 2007). A atividade de teste de software é uma atividade de VV&T dinâmica e uma das mais utilizadas.

Para Myers et al. (2004) teste é a atividade de executar um programa com o objetivo de encontrar defeitos. Basicamente, testar um programa ou modelo é executá-lo inserindo dados de teste (entradas) e comparando os valores de saída com os previstos na especificação. Ao obter uma saída não correspondente a esperada, tem-se uma falha no programa ou modelo em teste. Neste caso, deve-se realizar uma revisão a fim de identificar onde o defeito se encontra, atividade conhecida como depuração. O ideal é que a atividade de teste seja iniciada junto ao planejamento do sistema. De forma geral, ela deve incluir quatro fases (Maldonado, 1991): planejamento, projeto de casos de teste, execução dos testes e análise dos resultados dos testes. Para que os testes sejam realizados durante todo o processo de desenvolvimento do software são previstas três fases:

Teste de Unidade: o objetivo é testar as menores partes que constituem o sistema, como métodos, funções, procedimentos ou classes. Busca-se encontrar defeitos de programação, algoritmos incorretos ou mal implementados, estruturas de dados incorretas, ou outros tipos de defeitos, limitando-se a lógica interna e estruturas de dados dentro dos limites da unidade.

Teste de Integração: o foco principal é verificar as estruturas de comunicação entre as diversas unidades que compõem o sistema. As técnicas de projeto de casos de teste que exploram entradas e saídas são as mais utilizadas durante a integração, apesar das técnicas que exercitam caminhos específicos do programa poderem ser usadas para garantir a cobertura dos principais caminhos de controle (Pressman, 2005).

Teste de Sistema: o objetivo é verificar se as funcionalidades foram implementadas corretamente, se os requisitos não funcionais satisfazem a especificação e aspectos como correção, coerência e completude também são avaliados (Delamaro et al., 2007). O sistema é testado com outros sistemas de software e elementos de hardware que possivelmente estariam utilizando-o após sua liberação.

Teoricamente, os testes deveriam submeter o programa a todas as entradas possíveis para então verificar se as saídas corresponderiam ao esperado, nesse caso estaria sendo realizado um teste exaustivo. No entanto, o domínio de entrada muitas vezes grande, ou até infinito, impossibilita essa ação, limitando a execução dos testes a algumas entradas do domínio. Em decorrência dessa limitação não é permitido em geral afirmar a corretude de um programa (Delamaro et al., 2007). A fim de aumentar a abrangência dos testes no escopo de um sistema foram criadas as técnicas e critérios de teste.

Os critérios de teste são "regras" utilizadas para identificar os diferentes subdomínios do conjunto de entrada de um programa. Um subdomínio compreende parte do domínio de entrada, no qual os elementos possuem as mesmas características. Para identificar os subdomínios é necessário satisfazer os requisitos de teste, como exemplo, executar uma determinada estrutura do programa. Os dados de teste que satisfazem um mesmo requisito pertencem ao mesmo subdomínio. Os critérios de teste permitem criar diferentes regras que determinam diferentes subdomínios. Podem ser identificados três tipos de critérios: funcionais, estruturais e baseados em erros. A diferença entre essas técnicas é a origem da informação utilizada para estabelecer os subdomínios e construir os conjuntos de casos de teste.

2.2.1 Técnica de Teste Funcional

Utilizando essa técnica considera-se o sistema uma caixa preta e os dados de entrada são gerados com base na especificação do sistema. Na técnica funcional os detalhes de implementação não são considerados, pois o objetivo é testar as funções do sistema. Como todos os critérios da técnica funcional baseiam-se na especificação, é fundamental que esta esteja correta e bem escrita para tornar possível a geração de casos de teste efetivos em detectar os defeitos que possam existir. Uma grande vantagem dos testes funcionais é que eles podem ser aplicados em todas as fases e em qualquer programa, pois não são avaliados os detalhes de implementação do sistema. Uma limitação encontrada pelos critérios dessa técnica é a não garantia de que partes críticas do sistema sejam testadas. Alguns critérios dessa técnica são: Particionamento de Equivalência, Análise do Valor Limite, Grafo Causa-Efeito, *Error-Guessing*, Teste Funcional Sistemático e Método de Partição-Categoria (Linkman et al., 2003; Myers et al., 2004; Ostrand e Balcer, 1988).

2.2.2 Técnica de Teste Estrutural

Também conhecida como teste caixa branca. Os casos de teste são gerados com base na implementação do software, com o objetivo de avaliar se a estrutura interna da unidade está correta. No geral, a maioria dos critérios dessa técnica utiliza uma representação

do programa chamada Grafo de Fluxo de Controle (GFC), composto por nós que representam conjuntos de comandos executados sequencialmente e arestas que representam transferências de controle entre comandos (Myers et al., 2004). Dada a importância dos critérios estruturais para a proposta apresentada nesse trabalho, estes serão explicados em mais detalhes.

Um programa P pode ser descrito como um GFC ($G = (N, E, s)$), em que N representa o conjunto de nós, E o conjunto de arcos e s o nó de entrada. Um caminho pode ser representado no grafo por uma sequência finita de nós $(n_1, n_2, ..., n_k)$, $k \geq 2$, tal que existe um arco de n_i para $n_i + 1$, com $i = 1, 2, ..., K - 1$. Com exceção do primeiro e do último nó que compõem um caminho, se os demais nós são distintos então o caminho é "simples", porém se todos os nós são distintos, o caminho é "livre de laço". Se o primeiro nó de um caminho é o nó de entrada e o último nó é um nó de saída do grafo, então o caminho é "completo".

Uma extensão desse grafo é denominada Grafo Def-Uso, em que são acrescentadas informações sobre definição e uso de variáveis nos nós e arcos, representando o fluxo de dados do programa. Quando um valor é armazenado em uma posição de memória ocorre uma definição de variável. Num programa isso ocorre se a variável: encontra-se do lado esquerdo de um comando de atribuição, em um comando de entrada, ou em chamadas de procedimentos como parâmetro de saída. A passagem de valores entre procedimentos por meio de parâmetros pode ser por valor, por nome ou por referência (Ghezzi e Jazayeri, 1987). Num programa, uma variável passada por nome ou por referência é um parâmetro de saída. Uma definição numa chamada de procedimento é um definição realizada por referência. Quando não é possível a uma variável acesso a seu valor ou a definição de localização da variável não se encontra mais na memória, a variável está indefinida (Maldonado, 1991).

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

Para uma variável x definida em um nó n_i com uso no nó n_j ou em um arco que chega em n_j, se há um caminho $(n_i, n_1, ..., n_m, n_j)$, com $m \geq 0$ que não possua definição de x nos nós $n_1, ..., n_m$, esse é um "caminho livre de definição" com relação a x do nó n_i ao nó n_j e do nó n_i ao arco (n_m, n_j) (Maldonado, 1991).

```
/*****************************************************************************
Identifier.c
ESPECIFICACAO: O programa deve determinar se um identificador eh ou nao valido em 'Silly
Pascal' (uma estranha variante do Pascal). Um identificador valido deve comecar com uma
letra e conter apenas letras ou digitos. Alem disso, deve ter no minimo 1 caractere e no
maximo 6 caracteres de comprimento
*****************************************************************************/

            #include <stdio.h>
            main ()
/*  1 */    {
/*  1 */        char  achar;
/*  1 */        int length, valid_id;
/*  1 */        length = 0;
/*  1 */        valid_id = 1;
/*  1 */        printf ("Identificador: ");
/*  1 */        achar = fgetc (stdin);
/*  1 */        valid_id = valid_s(achar);
/*  1 */        if(valid_id)
/*  2 */        {
/*  2 */            length = 1;
/*  2 */        }
/*  3 */        achar = fgetc (stdin);
/*  4 */        while(achar != '\n')
/*  5 */        {
/*  5 */            if(!(valid_f(achar)))
/*  6 */            {
/*  6 */                valid_id = 0;
/*  6 */            }
/*  7 */            length++;
/*  7 */            achar = fgetc (stdin);
/*  7 */        }
/*  8 */        if(valid_id &&
                  (length >= 1) && (length < 6))
/*  9 */        {
/*  9 */            printf ("Valido\n");
/*  9 */        }
/* 10 */        else
/* 10 */        {
/* 10 */            printf ("Invalid\n");
/* 10 */        }
/* 11 */    }

                int valid_s(char ch)
/* 1 */         {
/* 1 */            if(((ch >= 'A') &&
                       (ch <= 'Z')) ||
                      ((ch >= 'a') &&
                       (ch <= 'z')))
/* 2 */            {
/* 2 */                return (1);
/* 2 */            }
/* 3 */            else
/* 3 */            {
/* 3 */                return (0);
/* 3 */            }
/* 4 */         }

                int valid_f(char ch)
/* 1 */         {
/* 1 */             if(((ch >= 'A') &&
                        (ch <= 'Z')) ||
                       ((ch >= 'a') &&
                        (ch <= 'z')) ||
                       ((ch >= '0') &&
                        (ch <= '9')))
/* 2 */             {
/* 2 */                 return (1);
/* 2 */             }
/* 3 */             else
/* 3 */             {
/* 3 */                 return (0);
/* 3 */             }
/* 4 */         }
```

Figura 2.1: Código do programa Identifier

Um exemplo retirado de Maldonado et al. (2004) pode ser usado para ilustrar a técnica de teste estrutural. O código do exemplo é apresentado na Figura 2.1 e o GFC para a função Main do programa Identifier gerada pela ferramenta *ViewGraph* (Vilela et al., 1997) é apresentado na Figura 2.2. O programa Identifier determina a validade ou não de um identificador (um identificador válido deve começar com uma letra e conter apenas letras ou dígitos e possuir no mínimo 1 e no máximo 6 caracteres de comprimento). No exemplo apresentado o programa contém um defeito.

No exemplo da Figura 2.1 o primeiro bloco de comandos que forma um nó no grafo compreende os comandos antecedidos pela marcação 1. O segundo bloco compreende os comandos com marcação 2 e assim por diante. Observando as Figuras 2.1 e 2.2, observa-se que o caminho $(2, 3, 4, 5, 6, 7)$ é um caminho simples e livre de laços e que o caminho $(1, 2, 3, 4, 5, 7, 4, 8, 9, 11)$ é um caminho completo. O caminho $(6, 7, 4, 5, 7, 4, 8)$ é um caminho livre de definição com relação a variável *valid_id* pois ela pode ter sido definida no nó n_6 e não ter ocorrido nenhuma redefinição dessa variável até seu uso no nó

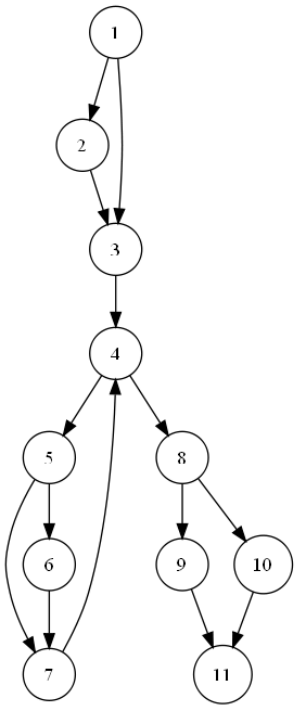

Figura 2.2: GFC do programa Identifier

n_8. O caminho $(6, 7, 4, 5, 7, 4, 8, 9)$ é não executável, porque a variável *valid_id* é redefinida no nó n_6 com o valor zero, tornando impossível satisfazer a condição no nó n_8 e executar o nó n_9. Qualquer outro caminho completo que o inclua também será não executável, pois não existe um dado de entrada que leve a execução desse caminho (Maldonado et al., 2004).

Os casos de teste visam a percorrer todos os caminhos internos possíveis da unidade, buscando descobrir defeitos no uso de variáveis, laços e condições. Um problema do teste estrutural é a presença de elementos requeridos não executáveis que impossibilitam testar em 100% a unidade. Os critérios da técnica estrutural são classificados com base no fluxo de controle, na complexidade e no fluxo de dados.

Critérios Baseados no Fluxo de Controle - utilizam o GFC e características de controle da execução do programa, como comandos ou desvios, para determinar as estruturas que devem ser testadas. Os critérios mais conhecidos dessa classe são: Todos-Nos (exige a execução de cada comando pelo menos uma vez), Todas-Arestas (requer a execução de cada desvio do fluxo de controle do programa pelo menos uma vez) e Todos-Caminhos (requer a execução de todos os caminhos possíveis do programa).

Critérios Baseados na Complexidade - utilizam informações sobre a complexidade do programa para gerar os requisitos de teste. Como critério dessa classe tem-se o critério de McCabe (ou teste de caminho básico) que faz uso da complexidade ciclomática

(métrica de software que proporciona uma medida quantitativa da complexidade lógica de um programa) do GFC para gerar os requisitos de teste (Delamaro et al., 2007). A complexidade ciclomática fornece um limite máximo para a quantidade de caminhos linearmente independentes (caminho que adiciona pelo menos um novo conjunto de comandos de processamento ou uma nova condição) que formam o conjunto básico do programa e fornecem um limite superior para o número de testes que deve ser conduzido para garantir que todas as instruções sejam executadas ao menos uma vez (Pressman, 2005).

Critérios Baseados no Fluxo de Dados - selecionam caminhos de um programa com base na localização das definições e usos das variáveis (Pressman, 2005), características não exploradas pelos critérios baseados no fluxo de controle. Em Rapps e Weyuker (1985) é apresentada uma motivação para utilização desses critérios, segundo a qual, como não se deve considerar suficientemente testado um programa se todos os seus comandos não foram exercitados, também não se deve considerar suficientemente testado um programa se todos os resultados computacionais não tiverem sido usados ao menos uma vez. Exemplos de critérios baseados no fluxo de dados são:

- **critérios de Rapps e Weyuker** (Rapps e Weyuker, 1985): em que os mais conhecidos são: Todas-Definições, requer que cada definição de variável seja exercitada pelo menos uma vez, por p-uso ou c-uso; Todos-Usos, requer que todas associações entre uma definição de variável e seus subsequentes usos sejam exercitados pelos casos de teste, por pelo menos um caminho livre de definição; Todos-du-Caminhos requer que toda associação entre uma definição de variável e subsequentes p-usos ou c-usos dessa variável sejam exercitados por todos os caminhos livres de definição e livres de laço que cubram essa associação.

- **Potenciais-Usos** (Maldonado, 1991): requerem associações independentemente da ocorrência explícita de uma referência ou uso a uma determinada definição. Se puder existir um uso dessa definição, um potencial uso e uma potencial associação é requerida. Os critérios que fazem parte da família de critérios Potenciais-Usos são: Todos-Potenciais-Usos, Todos-Potenciais-Usos/du e Todos-Potenciais-du-Caminhos. Em geral, requerem um número pequeno de casos de teste e nenhum outro critério baseado em fluxo de dados inclui os critérios potenciais-usos.

2.2.3 Técnica de Teste Baseado em Erros

Nessa técnica são utilizados os defeitos mais comuns que ocorrem durante o processo de desenvolvimento de software para derivar os requisitos de teste. O objetivo é construir casos de teste que revelem a presença ou ausência de defeitos específicos (DeMillo, 1986).

O teste de Mutação ou Análise de Mutantes é um critério da técnica baseada em erros, em que o artefato (programa ou especificação) em teste é alterado com a utilização dos operadores de mutação, como se "defeitos" estivessem sendo inseridos. Os artefatos gerados com as mudanças modeladas pelos operadores de mutação são chamados mutantes.

Na fase de execução dos testes, casos de teste são gerados para cobrir os subdomínios modelados pelos operadores de mutação. O objetivo é obter saídas distintas para o programa original e para o programa mutante. Caso a saída obtida do programa mutante seja diferente da obtida do programa original o mutante deve ser descartado, ou "morto", pois, o caso de teste foi capaz de expor a diferença entre o programa original e o programa mutante, significando que o defeito modelado pelo mutante não está presente no programa. Caso as saídas sejam iguais, duas situações podem ter ocorrido: ou o caso de teste não foi suficiente para identificar a diferença ente os dois programas ou os programas são equivalentes. Os mutantes equivalentes geram a mesma saída do programa original para qualquer valor do domínio de entrada e devem ser identificados e marcados como equivalentes. Um defeito é identificado quando um mutante que não é equivalente ao programa original obtém a saída correta e esperada, nesse caso, o mutante é denominado "error-revealing".

2.3 Programas Concorrentes

Os programas sequenciais possuem instruções executadas sequencialmente, de maneira que enquanto uma instrução está sendo executada há subutilização de tempo e de hardware, pois, outras instruções poderiam estar sendo executadas no mesmo espaço de tempo, aproveitando recursos ociosos do processador. A programação concorrente tem como objetivo otimizar o desempenho dessas aplicações, permitindo uma melhor utilização dos recursos.

Uma aplicação concorrente é formada por vários processos que interagem entre si para resolver um problema. Um processo é um programa em execução, formado pelo programa executável, seus dados, o contador de instruções, os registradores e todas as informações necessárias à sua execução (Tanenbaum, 2001). A concorrência pode existir tanto em sistemas com único processador como em sistemas multiprocessadores (com mais de um processador). Em sistemas monoprocessados existirá a falsa impressão de que os processos estão sendo executados ao mesmo tempo, fato que caracteriza o pseudo-paralelismo. Para que os processos sejam executados em paralelo é necessário que estes estejam sendo executados no mesmo instante de tempo, fato que caracteriza o paralelismo real (Souza et al., 2007b). A concorrência existe, a partir do momento em que dois ou mais proces-

sos iniciaram sua execução, mas não terminaram (Almasi e Gottlieb, 1994). Para existir concorrência é necessário que os processos envolvidos comuniquem-se e sincronizem-se.

A comunicação permite a um processo influenciar na execução de outros processos. Para que dois processos se comuniquem, um deve executar uma ação que o outro perceba, como exemplo, alterar o valor de uma variável ou enviar uma mensagem. Já a sincronização pode ser vista como o conjunto de restrições na ordenação de eventos. O programador pode empregar mecanismos de sincronização para atrasar a execução de um processo obedecendo determinadas restrições (Andrews e Schneider, 1983). A sincronização controla a ordem de execução de cada processo sobre os dados e recursos compartilhados.

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

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

Existem ferramentas capazes de ativar mecanismos que controlam processos concorrentes possibilitando desenvolver programas paralelos. Em Almasi e Gottlieb (1994) são definidas três ferramentas para construção de programas paralelos: ambientes de paralelização automática, no qual o compilador paralelizador aceita o programa sequencial e gera um programa paralelo; linguagens de programação concorrentes, que possuem comandos específicos aos programas concorrentes e; extensões para linguagens sequenciais, que são bibliotecas de comunicação que permitem adicionar comandos que não existem na linguagem sequencial.

Em Foster (1995) são definidos dois tipos de paradigmas para comunicação e sincronização entre processos concorrentes:

- Memória compartilhada - as tarefas compartilham uma memória comum em que elas lêem e escrevem. Mecanismos como semáforos, *busy-waiting* ou monitores estabelecem a ordem de execução dos processos e controlam o acesso às regiões críticas. Exemplos desse paradigma são: *Threads, Shared Memory, Ada e OpenMP.*

- Passagem de mensagem - compõe-se de bibliotecas de comunicação que permitem a criação de programas paralelos. O programa é escrito em uma linguagem sequencial, como C ou $Fortran$ que é estendida por meio de uma biblioteca que inclui funções para troca de mensagens entre os processos. Existem diferentes versões de ambientes de passagem de mensagens, como exemplos têm-se o PVM (*Parallel Virtual Machine*) e o MPI (*Message-Passing Interface*).

2.3.1 O Padrão MPI

O padrão MPI (*Message-Passing Interface*) foi definido por um fórum composto por universidades, empresas e institutos de diversos países que necessitavam de um padrão de interface para ambientes de passagem de mensagem. O MPI especifica sintaxe e semântica de uma biblioteca de funções disponíveis ao programador de aplicações paralelas nas linguagens sequenciais C, $Fortran$ e $C++$. Porém, a maneira como esta especificação deve ser implementada não está definida.

O MPI tem se tornado a mais popular biblioteca padrão para passagem de mensagem (Quinn, 2003). Ela permite ao programador controlar a utilização de memória, feita de forma explícita, possibilitando otimizar o desempenho das aplicações. Esse modelo de programação é considerado não-determinístico, pois a ordem de chegada das mensagens enviadas por dois processos A e B para um terceiro processo C não é conhecida. Entretanto, o MPI garante que duas mensagens enviadas de um processo A para um processo B chegarão na ordem de envio. É responsabilidade do programador assegurar o determinismo da computação quando este é necessário.

No momento em que um processo está sendo criado, o ambiente MPI atribui a cada processo um *id* (*rank*), armazenado como um inteiro, e um grupo de comunicação. O *rank* é um tipo de identificador de processo, que é usado para especificar a origem e o destino das mensagens. A comunicação entre processos acontece ponto-a-ponto (utilizando operações *send* e *receive*) ou por comunicação coletiva (utilizando operações como *broadcast, gather, barrier*, dentre outras). No padrão MPI uma computação é formada por um ou mais processos que se comunicam por sub-rotinas de envio e recebimento de

mensagens. A biblioteca especificada pelo MPI possui centenas de funções, disponíveis em vários repositórios.

A Figura 2.3 apresenta o código de um programa MPI que ilustra o não determinismo (Hausen, 2005). A forma de comunicação entre os processos é ponto-a-ponto. O MPI é inicializado pela função MPI_Init. A função MPI_Comm_rank (linha 7) obtém o número do processo que a invocou e o atribui à variável $myrank$. Supondo que sejam criados três processos instanciando esse programa, cada processo executaria o mesmo código, porém a estrutura $if/else$ determinaria os pontos do código a serem executados por cada processo. Inicialmente são criadas duas variáveis, $myrank$ e o vetor $data$, além da estrutura $status$ do $mpi.h$. Durante a execução o primeiro if selecionaria o processo 0 ao comparar o $rank$, nesse caso o primeiro dado do vetor $data$ receberia o valor 20.0 e em seguida uma mensagem seria enviada por meio do comando MPI_Send, com os seguintes parâmetros (linha 14): o endereço do *buffer* de envio da mensagem, a quantidade de dados a serem enviados, o tipo de dado a ser enviado, o $rank$ do processo destino, uma tag para a mensagem e o comunicador. O segundo if selecionaria o processo com rank 1 que adicionaria à primeira posição do vetor data o valor 30.0 e em seguida enviaria por meio de MPI_Send uma mensagem.

```
#include <mpi.h>
int main(int argc, char *argv[]) {
    int myrank;
    float data[2];
5   MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0) {
        data[0] = 20.0;
10      MPI_Send(data, 1, MPI_FLOAT, 2, 0, MPI_COMM_WORLD);
    }
    else if (myrank == 1) {
        data[0] = 30.0;
        MPI_Send(data, 1, MPI_FLOAT, 2, 0, MPI_COMM_WORLD);
15  }
    else if (myrank == 2) {
        MPI_Recv(&data[0], 1, MPI_FLOAT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
            &status);
        MPI_Recv(&data[1], 1, MPI_FLOAT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
            &status);
        printf("diferença = %.1f \n", data[0] - data[1]);
20  }
    MPI_Finalize();
    return 0;
}
```

Figura 2.3: Código de um programa exemplo ilustrando o não determinismo

Por fim, o terceiro processo de rank 2 seria selecionado pelo terceiro if e receberia por meio do MPI_Recv dois dados. Os parâmetros do MPI_Recv são: endereço de posição do vetor que receberá o dado, a quantidade de dados a serem enviados, o tipo de dado

enviado, processo do qual receberá a mensagem (nesse caso de qualquer processo), *tag* para a mensagem, comunicador utilizado e *status* do processo. Como um dos parâmetros do MPI_Recv não especificou de qual processo o dado seria recebido, não está prevista qual mensagem chegará primeiro, estando essa ação não determinística. O comando *printf* imprime o valor da diferença entre os valores recebidos. Nesse caso, a ordem de chegada das mensagens influenciará no resultado da diferença a ser impressa. O processo P^2, cuja variável $myrank = 2$ espera receber dados de quaisquer fontes com a primitiva MPI_recv. O não determinismo provoca a não garantia sobre o conteúdo do vetor *data* no processo P^2 após execução dos *receives*. Nesse exemplo, se P^2 sincronizar-se primeiro com P^0 e depois com P^1, o programa terá como saída "diferença = -10.0", porém, se P^2 sincronizar-se primeiro com P^1 e depois com P^0, o programa terá como saída "diferença = 10.0". Por fim o comando $MPI\_Finalize$ finaliza o ambiente de execução MPI.

2.4 Engenharia de Software Experimental

O processo de experimentação pode ser visto como base do processo científico. Os experimentos verificam as teorias, fornecendo conclusões obtidas por meio de avaliações e comparações. Há quatro métodos de experimentação usados no campo da Engenharia de Software (Basili, 1993):

- **Método científico**: um modelo do mundo real é construído com base na observação, como exemplo, um modelo de simulação.
- **Método de engenharia**: são estudadas as soluções atuais a fim de que sejam propostas mudanças que permitam sua evolução.
- **Método empírico**: um modelo é proposto e evoluído por meio de estudos teóricos, como exemplos estudos de caso e experimentos.
- **Método analítico**: uma teoria formal é proposta e comparada com observações empíricas.

Cada um desses métodos são mais aplicados em determinadas situações. O método analítico é usado por áreas mais formais da engenharia elétrica e ciência da computação. O método científico é mais aplicado em áreas como simulação de redes com vistas a evolução de desempenho. Porém, a simulação pode ser usada em outros contextos. Como o método científico, o método de engenharia é provavelmente o mais usado na indústria. Os estudos empíricos têm sido tradicionalmente usados nas ciências sociais e psicologia, que se assemelham com a Engenharia de Software por considerarem o estudo do fator

humano. Na Engenharia de Software o fator humano domina o processo, dado que o desenvolvimento de software é fruto do trabalho humano.

Na Engenharia de Software os objetivos da condução de experimentos são a caracterização, avaliação, previsão, controle e melhoria a respeito de produtos, processos, recursos, modelos e outros resultados e atividades do processo de desenvolvimento de software (Travassos, 2002). Além de avaliar essas técnicas da Engenharia de Software, a experimentação permite contribuir com a evolução, previsão, compreensão, controle e aperfeiçoar o processo de desenvolvimento de software e produtos (Basili et al., 1986). Os experimentos em Engenharia de Software são quasi-experimentos, pois nem sempre é possível prover aleatorização, um conjunto representativo de sujeitos do mundo real, dentre outras dificuldades existentes nesse contexto.

2.4.1 O Processo Experimental

A experimentação é um dos estudos primários, assim como estudos de caso, *surveys* e pesquisa ação (Sjoberg et al., 2007). Ela pode ser vista como um processo sistemático composto por fases, subprocessos e produtos gerados ao final de cada fase. Caracteriza-se por um processo que inicia-se na definição do experimento passando pelo planejamento, operação, análise e interpretação até a apresentação e empacotamento (Wohlin et al., 2000). Essas fases compõem o processo experimental ilustrado na Figura 2.4.

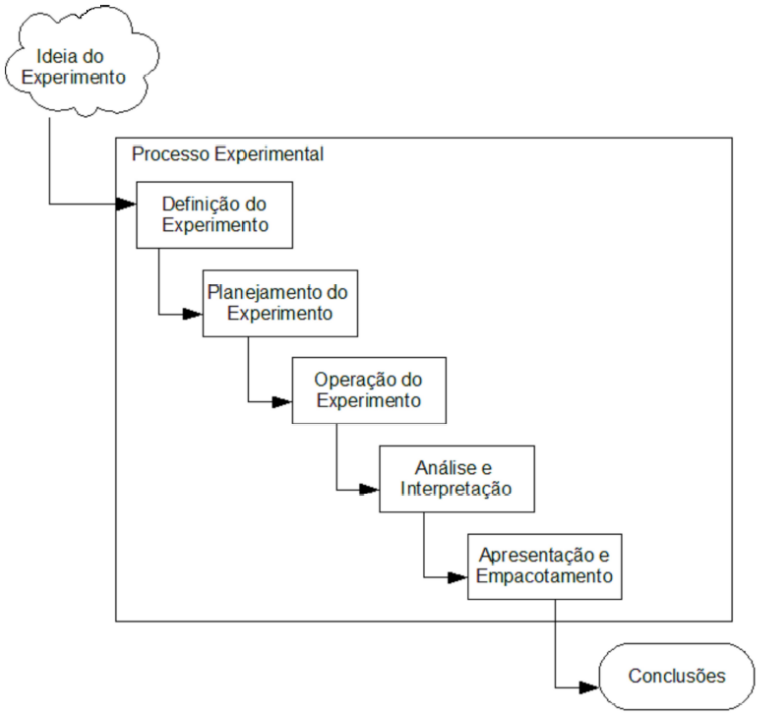

Figura 2.4: O processo de experimentação e suas fases (Wohlin et al., 2000)

O processo experimental tem início na definição. Nessa fase, são definidos aspectos gerais e específicos do experimento, relacionados ao problema que será tratado, objetivos e metas do estudo. Devem ser respondidas questões como: qual o objeto do estudo? (o que será estudado); qual o propósito do experimento? (intenção do estudo); qual será o foco de qualidade? (principal aspecto de qualidade que será estudado); qual perspectiva usar? (ponto de vista em que os resultados serão interpretados) e qual o contexto? (ambiente no qual o experimento será executado). Com base nas respostas a essas questões define-se o planejamento do experimento.

No planejamento o projeto do experimento é mais detalhado, incluindo também a avaliação dos riscos. São definidos o contexto, as hipóteses, as variáveis, os participantes, o projeto experimental, a instrumentação e a avaliação de validade do projeto experimental.

Três atividades compõem a fase de operação: preparação, execução e validação de dados. Durante a preparação o material que será utilizado no experimento é organizado, os participantes são informados a respeito do objetivo da realização do experimento e o consentimento de participação é preenchido pelos participantes. Durante a execução ocorre a condução do experimento. As tarefas previstas no planejamento são executadas e os dados são coletados. A validação dos dados é necessária para checar se estes foram coletados corretamente.

A fase de análise e interpretação recebe como entrada os dados coletados para que estes possam ser analisados e interpretados usando uma análise estatística. Pode ocorrer uma redução do conjunto de dados para remover informações (dados) que possam prejudicar a análise, ou a fim de reduzir o conjunto inicial para que a análise seja facilitada. Os resultados gerados pela análise e interpretação são usados pelo experimentador na tentativa de rejeitar a hipótese nula (teste de hipóteses) e confirmar a hipótese alternativa.

Nas fases de apresentação e empacotamento os resultados obtidos no experimento são apresentados e empacotados, por meio de documentação (artigos de pesquisa), pacotes de laboratório ou ainda formando uma base de conhecimento.

2.4.2 Experimentos em Teste de Software

Em razão da diversidade de técnicas de teste, existem muitos critérios que podem ser utilizados na execução de testes de um determinado sistema que se deseja testar. No área de teste de software, a abordagem empírica envolve coletar estatísticas sobre a frequência relativa com que as estratégias de teste obtém sucesso na detecção de erros em uma coleção de programas (Howden, 1978). Estudos experimentais podem ser conduzidos para avaliação de técnicas de teste, utilização de ferramentas, avaliação ou comparação de critérios de teste.

Alguns fatores podem ser utilizados para avaliação de critérios de teste em estudos experimentais, como: custo, eficácia e dificuldade de satisfação *(strength)*. O custo de um critério refere-se ao "trabalho" necessário para satisfazê-lo, e pode ser medido em função do número de casos de teste necessários para satisfazê-lo. Eficácia refere-se à capacidade que um critério possui em detectar um maior número de erros em relação a outro critério. Dificuldade de satisfação relaciona-se à probabilidade de satisfazer-se um critério já tendo sido satisfeito outro critério (Wong, 1993). A realização de estudos experimentais sobre critérios de teste é motivada pelas seguintes questões (Souza et al., 2007a):

- Dificuldade de satisfação entre dois critérios de teste;
- Custo de um critério de teste;
- Eficácia de um critério em revelar erros num programa;
- Redução de custo de um critério sem interferência na eficácia do critério;
- Consequências em reduzir o tamanho de um conjunto de casos de teste na eficácia do critério.

Alguns experimentos foram conduzidos para avaliar na prática aspectos dos critérios estruturais, cuja complexidade é exponencial. Em Weyuker (1990) são apresentados resultados de um estudo experimental conduzido para avaliar na prática a complexidade dos critérios baseados em fluxo de dados. Os resultados desse experimento mostram que os critérios de fluxo de dados são factíveis na prática. O mesmo *benchmark* usado no experimento apresentado em Weyuker (1990) foi utilizado para o primeiro experimento com os critérios Potenciais-Usos (Maldonado et al., 1992). Como resultados desse experimento observou-se que os critérios Potenciais-Usos, na prática, são factíveis e demandam um número de casos de teste relativamente pequeno.

No contexto de teste baseado em erros alguns estudos experimentais foram conduzidos, como em Mathur e Wong (1994) em que foi realizado um estudo para comparar a adequação de conjunto de casos de teste em relação aos critérios Análise de Mutantes e Todos-Usos, com o objetivo de verificar a dificuldade de satisfação entre os dois critérios, bem como custos. Como resultados do experimento os conjuntos adequados ao critério Análise de Mutantes também mostraram-se Todos-Usos-adequados, não sendo o inverso verdadeiro, possibilitando concluir que é mais difícil satisfazer o critério Análise de Mutantes que o critério Todos-Usos, inferindo-se que o critério Análise de Mutantes inclui Todos-Usos.

Outros experimentos foram realizados sobre o teste baseado em erros, com os objetivos de: avaliar e comparar critérios dessa técnica (Mathur e Wong, 1993), comparar critérios

do teste baseado em erros com critérios da técnica estrutural (Offutt et al., 1996b; Souza et al., 1997; Wong et al., 1995), avaliar estratégias para uso dos operadores de mutação (Offutt et al., 1996a), avaliar eficácia de operadores de mutação (Souza, 1996; Wong et al., 1994), avaliar custo do critério (Souza e Maldonado, 1997), dentre outros aspectos.

Em Linkman et al. (2003) é descrito um experimento usando critérios da técnica funcional: Particionamento de Equivalência, Análise do Valor Limite e Funcional Sistemático e o critério Análise de Mutantes, da técnica baseada em erros. Os objetivos foram avaliar a adequação do teste aleatório e funcional em relação ao critério Análise de Mutantes. O experimento indicou que os critérios de teste funcionais podem determinar um alto grau de cobertura do código-fonte do programa em teste, revelando um grande número de defeitos com baixo custo de aplicação. Outros aspectos do teste funcional também têm sido explorados em experimentos relatados na literatura, com o objetivo de compará-lo com outras técnicas de teste e inspeção de software (Dória, 2001; Hetzel, 1976; Kamsties e Lott, 1995; Myers, 1978; Wood et al., 1997).

2.5 Considerações Finais

Neste capítulo foi apresentada uma visão geral sobre os principais conceitos envolvidos neste projeto. Inicialmente foram descritos os principais conceitos de teste, abordando as técnicas de teste (funcional, estrutural e baseada em erros). Em seguida foram apresentados conceitos sobre programas concorrentes, enfatizando os principais paradigmas de implementação e o padrão de passagem de mensagem MPI. Por fim, foram abordados conceitos sobre Engenharia de Software Experimental.

Os estudos experimentais aqui apresentados tiveram como objetivo explanar o que tem sido feito de experimentação em teste de software, pois nesta proposta de trabalho está incluída a realização de um estudo experimental. No próximo capítulo será apresentada uma visão sobre a aplicação de teste de software no contexto de programas concorrentes.

Capítulo

3

Teste de Programas Concorrentes

3.1 Considerações Iniciais

Neste capítulo são apresentados conceitos sobre o teste de programas concorrentes, o qual impõe desafios, tais como: o desenvolvimento de técnicas de análise estática apropriadas para esses programas, detecção de erros de sincronização, comunicação, de fluxo de dados e de *deadlock*, reproduzibilidade, geração de representação do programa concorrente com informações pertinentes ao teste, adaptação de critérios de teste sequenciais para programas concorrentes (Yang e Pollock, 1997).

Na Seção 3.1 são apresentados os principais conceitos sobre teste de programas concorrentes, incluindo um modelo para teste estrutural, critérios de teste e a ferramenta de teste ValiMPI. Na Seção 3.2 são descritos os trabalhos relacionados ao teste de programas concorrentes. Por fim, na Seção 3.3 são apresentadas as considerações finais deste capítulo.

3.2 Teste Estrutural de Programas Concorrentes

Uma das linhas de pesquisa em Engenharia de Software do ICMC/USP está relacionada ao teste de software de programas concorrentes. Nessa linha, têm sido investigados a definição de modelos, critérios e ferramentas de teste considerando informações sobre o fluxo de controle, dados e comunicação desses programas. Nesse contexto, Souza et al.

(2008b) definiram um modelo de teste com base em Yang e Chung (1992) que representa as principais características de programas concorrentes de passagem de mensagem. No modelo proposto é considerado um número fixo n e conhecido de processos criados na inicialização da aplicação concorrente. Cada processo pode executar diferentes computações, porém utilizando seu próprio espaço de memória. A comunicação entre os processos é feita por meio de dois mecanismos básicos: a comunicação ponto-a-ponto, na qual um processo pode enviar uma mensagem para outro usando primitivas como *send* e *receive* e a comunicação coletiva, na qual um processo pode enviar uma mensagem para todos os processos da aplicação (ou para um particular grupo de processos).

Para um conjunto de processos $Prog = p^0, p^1, ..., p^{n-1}$ que formam um programa paralelo, é construído um GFC para cada processo p, utilizando os mesmos conceitos adotados para programas sequenciais. Cada GFC^p representa o fluxo de controle de um processo p e possui dois nós especiais: o nó de entrada e o nó de saída, que representam o início e o término da execução de um processo p. Um nó n pode ou não estar associado a uma função de comunicação (*send* ou *receive*). As primitivas de comunicação são associadas pelas notações $send(p, k, t)$ ou $receive(p, k, t)$, significando que o processo p envia (ou recebe) uma mensagem t para o (ou do) processo k. Um GFCP construído para $Prog$ é formado pelos GFC^p (para $p = 0...n-1$) e pelas arestas de comunicação dos processos paralelos. N representa o conjunto de nós e E o conjunto de arestas do GFCP. E possui como subconjuntos: E_p, que são arestas de um mesmo processo e E_s que são as arestas interprocessos. Um nó i num processo p é representado pela notação n_i^p. Dois subconjuntos de N são definidos: N_s, que são os nós com funções de envio de mensagens e N_r são os nós que possuem recebimento de mensagens. Para cada $n_i^p \in N_s$ um conjunto R_i^p está associado contendo possíveis nós que recebem a mensagem enviada pelo nó n_i^p.

Um caminho π no GFC^p é chamado intraprocesso se ele não contém nenhuma aresta interprocesso, e é dado por uma sequência de nós $\pi = (n_1, n_2, ..., n_m)$, tal que $(n_i, n_{i+1}) \in E_i^p$. Um caminho π que contém pelo menos uma aresta interprocessos é dito um caminho interprocessos. Para programas escritos no ambiente de passagem de mensagem, além das definições de variáveis comuns aos programas sequenciais (Seção 2.3.2, Cap 2), um novo tipo de definição de variável é possível: a definição decorrente da execução de uma função de comunicação tal como uma primitiva *receive*. O conjunto de variáveis definidas em um nó N_i^p é dado por $def(n_i^p)$. O uso de uma variável x ocorre quando x é referenciada em uma expressão. No contexto de programas concorrentes podem ocorrer três formas de uso de variáveis:

- Uso computacional (*c-uso*): ocorre numa computação, relacionado a um nó n_i^p do GFCP;

- Uso predicativo (*p-uso*): ocorre numa condição (predicado), associado a comandos de fluxo de controle, em arestas intraprocessos (n_i^p, n^p) do GFCP;

- Uso comunicacional (*s-uso*): ocorre em primitivas de sincronização (*send* e *receive*), em arestas interprocessos $(n_i^{p_1}, n_j^{p_2}) \in E_s$.

Com base nessas informações três tipos de associações são requeridas pelos critérios estruturais aplicados a programas concorrentes:

- Associação *c-uso*: dada pela tripla (n_i^p, n_j^p, x) em que $x \in def(n_i^p)$, e n_j^p possui um *c-uso* de x e existe um caminho livre de definição com relação a x de n_i^p para n_j^p;

- Associação *p-uso*: dada pela tripla $(n_i^p, (n_j^p, n_k^p), x)$ em que $x \in def(n_i^p)$, e (n_j^p, n_k^p) possui um *p-uso* de x e existe um caminho livre de definição com relação a x de n_i^p para (n_j^p, n_k^p);

- Associação *s-uso*: dada pela tripla $(n_i^p, (n_j^{p_1}, n_k^{p_2}), x)$ em que $x \in def(n_i^{p_1})$, e $(n_j^{p_1}, n_k^{p_2})$ possui um *s-uso* de x e existe um caminho livre de definição com relação a x de $n_i^{p_1}$ para $(n_j^{p_1}, n_k^{p_2})$.

As associações *p-uso* e *c-uso* são associações intraprocessos, isto é, a definição e o uso de x ocorrem num mesmo processo p_1. Já uma associação *s-uso* requer a participação de outro processo p_2, caracterizando assim uma associação interprocessos. Estas permitem identificar erros de envio e recebimento de mensagens. Outros tipos de associações interprocessos podem ser caracterizadas considerando associações *s-usos* e comunicação interprocessos:

- Associação *s-c-uso*: dada pela quíntupla $(n_i^{p_1}, (n_j^{p_1}, n_k^{p_2}), n_l^{p_2}, x^{p_1}, x^{p_2})$ tal que existe uma associação *s-uso* $(n_i^{p_1}, (n_j^{p_1}, n_k^{p_2}), x^{p_1})$ e uma associação *c-uso* $(n_k^{p_2}, (n_l^{p_2}, x^{p_2})$;

- Associação *s-p-uso*: dada pela quíntupla $(n_i^{p_1}, (n_j^{p_1}, n_k^{p_2}), (n_l^{p_2}, x_m^{p_2}), x^{p_1}, x^{p_2})$ tal que existe uma associação *s-uso* $(n_i^{p_1}, (n_j^{p_1}, n_k^{p_2}), x^{p_1})$ e uma associação *p-uso* $(n_k^{p_2}, (n_l^{p_2}, n_m^{p_2})$ $, x^{p_2})$.

Souza et al. (2008b) definiram duas famílias de critérios de teste estruturais para programas concorrentes em ambientes de passagem de mensagem: critérios baseados em fluxo de controle e no fluxo de comunicação e os critérios baseados em fluxo de dados e em passagem de mensagem.

Critérios baseados em fluxo de controle e no fluxo de comunicação

Cada GFC^p (para $p = 0..n-1$) pode ser testado individualmente utilizando-se os critérios todos-nos e todas-arestas, pois tratando-se da execução de um único processo, o teste iguala-se ao teste de programas sequenciais. No entanto, para testar a comunicação entre processos os seguintes critérios são definidos:

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

Definir os critérios todos-caminhos do GFC^p (para $p = 0..n$-1) e todos-caminhos do GFCP, equivale a executar todas as combinações possíveis dos caminhos nos GFC^p, porém esses critérios são em geral impraticáveis, devido ao problema da explosão de estados.

Critérios baseados em fluxo de dados e em passagem de mensagens

- **todas-defs**: requer que para cada nó n_i^p e cada x em $def(n_i^p)$ uma associação definição-uso (*c-uso* ou *p-uso*) com relação a x seja exercitada pelo conjunto de casos de teste;
- **todas-defs/s**: requer que para cada nó n_i^p e cada x em $def(n_i^p)$ uma associação *s-c-uso* ou *s-p-uso* com relação a x seja exercitada pelo conjunto de casos de teste. Se não existir associação *s-c-uso* ou *s-p-uso* com relação a x, é requerida qualquer outro tipo de associação intraprocesso em relação a x;
- **todos-c-usos**: requer que todas as associações *c-usos* sejam exercitadas pelo conjunto de casos de teste;
- **todos-p-usos**: requer que todas as associações *p-usos* sejam exercitadas pelo conjunto de casos de teste;

- **todos-s-usos**: requer que todas as associações *s-usos* sejam exercitadas pelo conjunto de casos de teste;
- **todos-s-c-usos**: requer que todas as associações *s-c-usos* sejam exercitadas pelo conjunto de casos de teste;
- **todos-s-p-usos**: requer que todas as associações *s-p-usos* sejam exercitadas pelo conjunto de casos de teste.

Um problema que ocorre nos programas sequenciais que também está presente nos programas concorrentes é a não executabilidade, que é inerente a esses critérios (Souza et al., 2007b). Os critérios de teste definidos anteriormente estão implementados na ferramenta ValiMPI, a qual será apresentada em mais detalhes.

3.2.1 A Ferramenta ValiMPI

A ferramenta ValiMPI (Hausen, 2005) testa programas escritos na linguagem de programação *C* que usam a biblioteca de passagem de mensagem MPI. Ela fornece funções para criar uma sessão de teste, salvar, executar dados de teste e evoluir a cobertura do teste para um dado critério. Atualmente os seguintes critérios estruturais estão implementados na ferramenta: todos-nos, todos-nos-r, todos-nos-s, todas-arestas, todas-arestas-s, todos-c-usos, todos-p-usos e todos-s-usos. A ferramenta é composta por 5 módulos principais que se comunicam por meio de arquivos. A interação entre os módulos é detalhada na Figura 3.1.

Vali-Inst - realiza a instrumentação do programa em teste e extração das informações de fluxo de controle e de dados. Esse módulo recebe como entrada o programa fonte original e a descrição semântica da instrumentação e gera como saída o programa instrumentado, os GFCs e informações sobre o fluxo de dados. A IDeL (*Instrumentation Description Language*) (Simão et al., 2003) realiza a instrumentação propriamente dita. No programa instrumentado algumas declarações são inseridas diferindo-o do programa original, como exemplos, a inclusão do comando *check_point* que identifica o rastro de execução, substituição de funções como: *MPI_Send, MPI_Isend, MPI_Recv e MPI_Irecv* por *ValiMPI_Send, ValiMPI_Isend, ValiMPI_Recv* e *ValiMPI_Irecv*, as chamadas de funções de teste e espera de requisições de envio ou recebimento de mensagens não-bloqueantes *MPI_Test* e *MPI_Wait* que são substituídas respectivamente por *ValiMPI_Test* e *ValiMPI_Wait*, funções de início e término do ambiente paralelo também têm suas versões modificadas implementadas, *ValiMPI_Init* e *ValiMPI_Finalize* respectivamente.

Todavia, o programa instrumentado mantém as chamadas aos nomes originais das funções. A substituição é feita em tempo de "link - edição" do executável por meio do

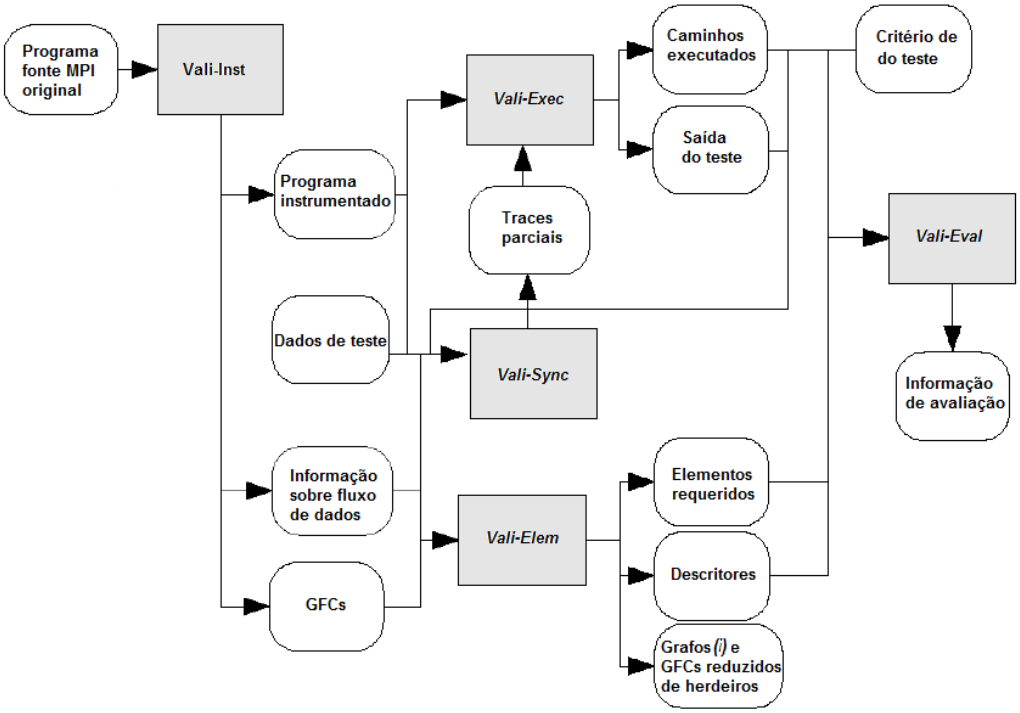

Figura 3.1: Arquitetura da ValiMPI

mecanismo de *profiling* descrito no padrão MPI 1.1, possibilitando as chamadas de início e término do ambiente paralelo estarem presentes em arquivos fonte não instrumentados. Essas mudanças não alteram a semântica do programa, servem para coletar informações necessárias ao teste. O módulo **Vali-Inst** também gera os grafos de fluxo, contendo informações sobre desvios, definição e uso de variáveis, e troca de mensagens. Por fim, os arquivos temporários são apagados e o programa instrumentado é identado para facilitar sua leitura por parte do usuário.

Vali-Elem - realiza a tarefa de geração dos elementos requeridos. Esse módulo recebe como entrada os GFCs, as informações de fluxo de dados e a distribuição das funções em teste entre os processos, gerando como saída os elementos requeridos pelos critérios baseados em fluxo de controle e fluxo de dados. Também são gerados descritores dos elementos requeridos, para isso são utilizados dois outros grafos: o grafo reduzido de herdeiros (Chusho, 1987) e o grafo (i), usado pela ferramenta de teste Poketool (Chaim, 1991).

No grafo reduzido de herdeiros todas as arestas são primitivas. O algoritmo de geração do grafo baseia-se nessa observação: quando uma aresta é executada num GFC existem outras arestas que são sempre executadas. Se todo caminho completo que inclui a aresta E_a sempre incluir a aresta E_b, então E_b é chamada de aresta herdeira de E_a e E_a é chamada ancestral de E_b, pois E_b herda informação de execução de E_a. O conceito

de aresta primitiva é então estabelecido em decorrência do conceito de aresta herdeira, sendo entendido como aresta primitiva a aresta que não é herdeira de nenhuma outra. Ao conceito de aresta primitiva foi adicionado o conceito de aresta de comunicação (aresta que liga dois processos em um GFCP). É possível minimizar o número de arestas requeridas pela ValiMPI utilizando os conceitos de aresta primitiva e de aresta de comunicação.

Um grafo(i) é construído para cada nó N_i que contenha definição de variável, sendo que, um dado nó N_k pertencerá a um grafo(i), se existir, pelo menos um caminho do nó N_i para N_k livre de definição com relação a pelo menos uma variável V_1 definida em N_i. Considerando $C(i, k)$ o conjunto de todos os caminhos livres de laço do nó N_i ao nó N_k no GFC, então N_k pertencerá a um grafo(i) e por conseguinte, todos os demais nós de um caminho C_1, se existir $C_1 \in C(N_i, N_k)$ e $V_1 \in def(N_i)$, tal que C_1 é livre de definição com relação a V_1. Dessa forma, um nó, ou aresta, do grafo pode dar origem a vários nós ou arestas, no grafo(i). Para evitar caminhos infinitos, causados por laços no GFC, em um mesmo caminho do grafo(i), apenas um nó pode conter mais de uma imagem, e sua imagem é o último nó do caminho. O grafo(i) é utilizado para estabelecer associações, definições e usos de variáveis e elementos requeridos por critérios baseados em fluxo de dados.

Para cada elemento requerido, **Vali-Elem** gera um descritor que será utilizado na avaliação (**Vali-Eval**). Um descritor é uma expressão regular que descreve um caminho que cobre um elemento requerido.

Vali-Exec - executa o programa instrumentado com os dados de teste (argumentos de linha de comando) fornecidos pelo usuário e gera como saída o traço de execução e as sequências de sincronizações para cada processo paralelo. Esse módulo é implementado por um *script* que inicia o ambiente paralelo, se necessário, e executa o programa em paralelo. O traço de execução é composto pelos caminhos interprocessos executados para cada processo. **Vali-Exec** armazena as entradas, as saídas geradas, o número de processos, o traço de execução de cada função por processo e a sequência de sincronizações para cada processo.

Durante a execução do programa, o usuário pode visualizar as saídas do **Vali-Exec** e do programa em teste a fim de determinar se a saída obtida é a esperada. Caso as saídas sejam diferentes, é identificado um erro que deve ser corrigido antes de prosseguir com o restante dos testes. O traço de execução também pode servir como auxílio à depuração. **Vali-Exec** também habilita a execução controlada do programa em teste, usada na reprodução de um caso de teste já executado, ou para tentar a cobertura de sincronizações não executadas. Nesse caso são utilizados dados de teste armazenados e a sequência de sincronizações executadas do caso de teste que se deseja reproduzir, essa coleta de dados

de sincronizações é feita durante a execução do programa em teste. Ao final da execução, a saída produzida é comparada com a saída anterior, gerando um aviso ao usuário em caso de divergência. Os caminhos executados gerados pelo **Vali-Exec** são utilizados como entrada para o módulo **Vali-Sync**. A execução controlada garante que duas execuções do programa paralelo com a mesma entrada produzirão os mesmos caminhos e as mesmas sequências de sincronizações.

Vali-Sync - obtém as sincronizações interprocessos entre as chamadas de *send* e *receive* do MPI e realiza a reexecução de cada uma dessas sincronizações de forma a testar o programa paralelo para erros temporais. O módulo utiliza arquivos gerados pela execução de outros módulos da ferramenta. Devido às limitações desse módulo, o arquivo *.c* instrumentado não deve apresentar tipo de retorno de função implícito, definição de variáveis em escopos menores do que o da função e definição da variável junto de sua declaração. A execução do módulo gera os arquivos de sincronização e faz a execução de cada uma dessas sincronizações. Os traces parciais gerados pela ferramenta são utilizados como entradas para o módulo **Vali-Exec**.

Vali-Eval - avalia a cobertura dos casos de teste. Esse módulo recebe como entrada um critério de teste selecionado pelo usuário, os descritores e elementos requeridos para o critério selecionado para os casos de teste executados pelo **Vali-Exec** e gera como saída informações sobre a cobertura do critério obtida pelos casos de teste. **Vali-Eval** implementa um autômato que possui como entrada o caminho executado e verifica quais os descritores estão em seus estados finais, pois, isso significa que os elementos requeridos descritos por aqueles descritores foram cobertos. Por meio do traço de execução dos casos de teste verificam-se quais os elementos requeridos pelo critério foram cobertos. O módulo fornece ao usuário o percentual de cobertura dos casos de teste, a lista dos elementos requeridos não cobertos e os cobertos acompanhada dos casos de teste que os cobriram.

Para ilustrar as definições apresentadas sobre critérios de teste e uso da ferrramenta ValiMPI, considere o programa GCD ilustrado na Figura 3.2. Esse programa calcula o máximo divisor comum entre 3 números, utilizando comunicação ponto-a-ponto em MPI. Para que o programa seja executado é necessário que sejam instanciados 4 processos. O ambiente MPI é inicializado pela função MPI_Init. A função MPI_Comm_rank obtém o número do processo que a invocou e o atribui à variável $myRank$.

Para avaliar o programa *gcd* utilizando a ferramenta ValiMPI, é necessário que todos os comandos descritos sejam executados na pasta onde estão o código fonte do programa *gcd*. Essa ordem de passos deve ser seguida:

```
*********************************************
Máximo divisor comum entre 3 números - Versão MPI
*********************************************
        #include <stdio.h>
        #include <stdlib.h>
        #include <mpi.h>
        #include "gcd.h"

        void Master(int, int, int);
        void Slave(int rank);

        int main(int argc, char** argv)
* 1 */ {
* 1 */    int myRank;
* 1 */    int x, y, z;
* 2 */    if (argc != 4)
* 14 */    {
* 14 */          puts("argc != 4");
* 14 */          return 1;
* 14 */    }
* 3 */    MPI_Init(&argc, &argv);
* 4 */    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
* 5 */    if (myRank == 0)
* 6 */    {
* 6 */          x = atoi(argv[1]);
* 7 */          y = atoi(argv[2]);
* 8 */          z = atoi(argv[3]);
* 9 */      Master(x, y, z);
* 10 */    }
* 11 */    else
* 12 */    {
* 12 */       Slave(myRank);
* 12 */    }
* 13 */    MPI_Finalize();
* 15 */    return 0;
* 15 */ }

                // processo Mestre
                void Master(int x, int y, int z)
* 1 */ {
* 1 */    int buf[3];
* 2 */    buf[0] = x;
* 3 */    buf[1] = y;
* 4 */    MPI_Send(buf, 2, MPI_INT, 1, 1, MPI_COMM_WORLD);
* 5 */    buf[0] = y;
* 6 */    buf[1] = z;
* 7 */    MPI_Send(buf, 2, MPI_INT, 2, 1, MPI_COMM_WORLD);
* 8 */    MPI_Recv(buf, 1, MPI_INT, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &status);
* 9 */    x = buf[0];
* 10 */    MPI_Recv(buf, 1, MPI_INT, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &status);
* 11 */    y = buf[0];
* 12 */    if (x > 1 && y > 1)
* 13 */    {
* 13 */       buf[0] = x;
* 14 */       buf[1] = y;
* 15 */       MPI_Send(buf, 2, MPI_INT, 3, 1, MPI_COMM_WORLD);
* 16 */       MPI_Recv(buf, 1, MPI_INT, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &status);
* 17 */       z = buf[0];
* 18 */    }
* 19 */    else
* 19 */    {
* 19 */       buf[0] = -1;
* 20 */       buf[1] = -1;
* 21 */       MPI_Send(buf, 2, MPI_INT, 3, 1, MPI_COMM_WORLD);
* 22 */       MPI_Recv(buf, 1, MPI_INT, 3, 2, MPI_COMM_WORLD, &status);
* 23 */       z = 1;
* 23 */    }
* 25 */    printf("resultado = %d\n", z);
* 26 */ }

                // Processo Escravo
                void Slave(int rank)
* 1 */ {
* 1 */    int buf[2];
* 2 */    MPI_Recv(buf, 2, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
* 4 */    while (buf[0] != buf[1])
* 5 */    {
* 5 */       if (buf[0] < buf[1])
* 6 */          buf[1] = buf[1] - buf[0];
* 8 */       else
* 9 */          buf[0] = buf[0] - buf[1];
* 10 */    }
* 11 */    MPI_Send(buf, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
* 12 */ }
```

Figura 3.2: Código do programa GCD

1. Utilizar o módulo **Vali-Inst** para instrumentar o programa e gerar os grafos de fluxo de controle e de dados. O seguinte comando realiza essa ação:
   vali_inst gcd.c
   como saída são gerados os arquivos *gcd.c_instrumentado.c* que é o código fonte instrumentado e os arquivos *C.main.dot*, *Master.dot* e *Slave.dot* que são os arquivos de descrição dos GFCs de cada função (*main, Master e Slave*) do código fonte. Na Figura 3.3 está ilustrado o grafo GFCP para o programa GCD, construído a partir dos arquivos *Master.dot* e *Slave.dot*, com representação de algumas arestas interprocessos.

2. Compilar o programa pelo *script vali_cc*, usando o seguinte comando:
   vali_cc gcd.c_instrumentado.c -o gcd
   Como saída é gerado o arquivo *gcd*, que é o executável do programa.

3. Utilizar o módulo **Vali-Elem** para gerar os elementos requeridos e descritores de critérios, é usado o seguinte comando, que para esse exemplo considerou o processo 0 como o principal e os processos $1, 2, 3$ como os que realizam as computações:
   vali_elem 4 "Master(0)" "Slave(1,2,3)"

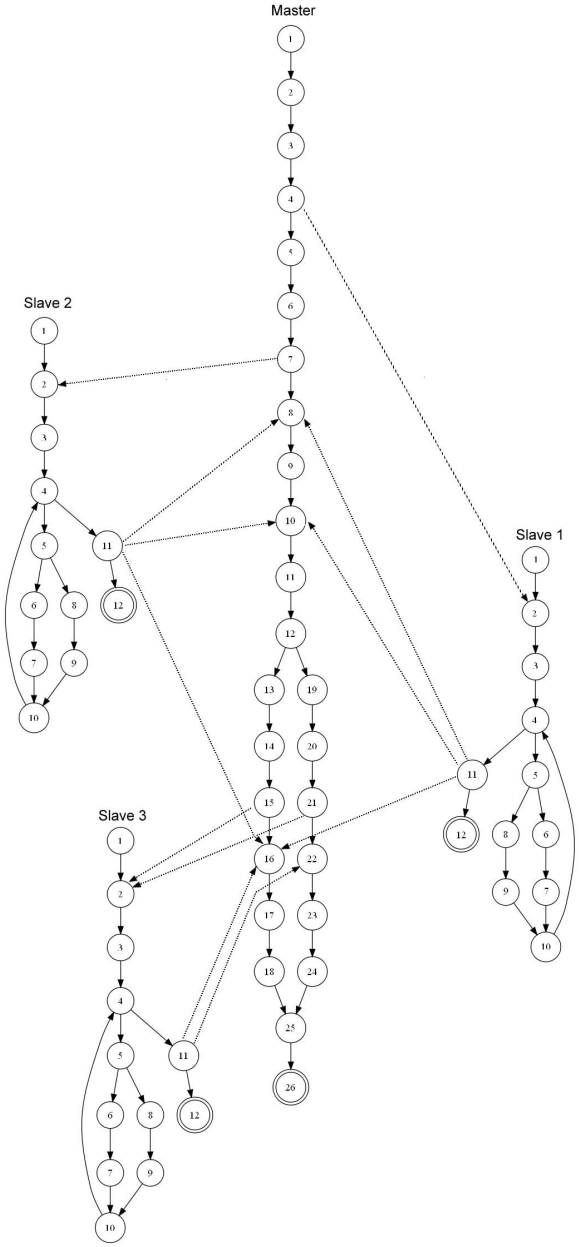

Figura 3.3: GFCP do programa GCD

Como saída são gerados os elementos requeridos e descritores no diretório *./valimpi/res/*. Na Figura 3.4 está ilustrado o arquivo que descreve os elementos requeridos gerados para o critério todas-arestas-s. Nesse caso, o elemento requerido 1 indica que a aresta entre o nó n_4 do processo P^0 e o nó n_2 do processo P^1 deve ser coberta, o elemento requerido 2 indica que a aresta entre o nó n_4 do processo P^0 e o nó n_2 do processo P^2 deve ser coberta, e assim por diante.

4. Para executar um determinado caso de teste é usado o módulo **Vali-Exec**. Para o comando ilustrado, é utilizada a entrada de teste $(12, 44, 36)$
   vali_exec 1 run 4 gcdMPI "12 44 36"

```
ELEMENTOS REQUERIDOS PARA O CRITERIO TODAS AS ARESTAS S

1) 4-0 2-1
2) 4-0 2-2
3) 4-0 2-3
4) 7-0 2-1
5) 7-0 2-2
6) 7-0 2-3
7) 15-0 2-1
8) 15-0 2-2
9) 15-0 2-3
10) 21-0 2-1
11) 21-0 2-2
12) 21-0 2-3
13) 11-1 8-0
14) 11-1 10-0
15) 11-1 16-0
16) 11-1 22-0
17) 11-1 2-2
18) 11-1 2-3
19) 11-2 8-0
20) 11-2 10-0
21) 11-2 16-0
22) 11-2 22-0
23) 11-2 2-1
24) 11-2 2-3
25) 11-3 8-0
26) 11-3 10-0
27) 11-3 16-0
28) 11-3 22-0
29) 11-3 2-1
30) 11-3 2-2
```

Figura 3.4: Elementos requeridos para o critério todas-arestas-s

Como saída o programa intrumentado é impresso na tela e os seguintes arquivos são criados no diretório *./valimpi/test_case0001*: *progname* (com o nome do executável), *stdout* (com o conteúdo da saída padrão), *stderr* (com o conteúdo da saída do erro), *args* (com os argumentos de entrada do programa), *stdin* (com o conteúdo da entrada padrão) e *trace.main.p*[0...3] (com os arquivos de caminho de execução de cada processo) e *seqsync.p*[0...3] (com os arquivos de sequência de sincronização de cada processo). Se a implementação LAM/MPI estiver disponível, é gerado o arquivo *xmpi.lamtr*, que serve para visualização da execução do programa.

5. Com todas as saídas necessárias já geradas é possível executar o **Vali-Sync** por meio do comando:
   vali_sync gcdMPI
   A execução do módulo gera os arquivos de sincronização e faz a execução de cada uma dessas sincronizações. São produzidos os arquivos *lsTrace* e *lsDot* que são usados durante a execução do módulo. No diretório *./valimpi/seq_syncs* são criados os arquivos *seq_syc.p*[0..3]_[0..6] que contêm as diferentes sincronizações possíveis encontradas pelo módulo.

6. Avaliar a cobertura obtida pelo caso de teste para um dado critério selecionado, por meio do módulo **Vali-Eval**. Essa análise é realizada com a utilização dos descritores, elementos requeridos e os caminhos executados pelo caso de teste para identificar quais elementos requeridos foram cobertos para um dado critério de teste. O escore de cobertura e a relação dos elementos cobertos e não cobertos para o critério de teste selecionado é fornecida como saída pela ferramenta. A Figura 3.5 ilustra a saída do **Vali-Eval** para o programa exemplo e a entrada de teste considerada.

```
-- ELEMENTOS REQUERIDOS COBERTOS --
1) 4-0 2-1,  coberto por valimpi/test_case0001
5) 7-0 2-2,  coberto por valimpi/test_case0001
9) 15-0 2-3,  coberto por valimpi/test_case0001
13) 11-1 8-0,  coberto por valimpi/test_case0001
20) 11-2 10-0,  coberto por valimpi/test_case0001
27) 11-3 16-0,  coberto por valimpi/test_case0001

-- ELEMENTOS REQUERIDOS NÃƒO COBERTOS --
2) 4-0 2-2
3) 4-0 2-3
4) 7-0 2-1
6) 7-0 2-3
7) 15-0 2-1
8) 15-0 2-2
10) 21-0 2-1
11) 21-0 2-2
12) 21-0 2-3
14) 11-1 10-0
15) 11-1 16-0
16) 11-1 22-0
17) 11-1 2-2
18) 11-1 2-3
19) 11-2 8-0
21) 11-2 16-0
22) 11-2 22-0
23) 11-2 2-1
24) 11-2 2-3
25) 11-3 8-0
26) 11-3 10-0
28) 11-3 22-0
29) 11-3 2-1
30) 11-3 2-2

Cobertura para o criterio todas-arestasS: 20.00%
```

Figura 3.5: Saída do ValiEval para a cobertura do critério todas-arestas-s

Nesse exemplo, a cobertura de 20% significa que a entrada de teste obteve essa cobertura para o critério todas-arestas-s. Para que o critério seja satisfeito é necessário que sejam inseridas e executadas outras entradas de teste por meio do **Vali-Exec** até que a cobertura chegue próximo de 100%, descontando os elementos não executáveis que eventualmente existem para esse critério, os quais não permitem serem exercitados.

3.3 Trabalhos Relacionados ao Teste de Programas Concorrentes

Parte dos trabalhos apresentados nesta seção é resultado de uma revisão sistemática e um mapeamento sistemático conduzidos no contexto de teste para programas concorrentes, porém por questão de espaço, não foram incluídos neste trabalho. A revisão sistemática conduzida está descrita em Brito et al. (2010) e o mapeamento sistemático identificou trabalhos sobre teste de programas concorrentes num contexto mais amplo, o qual foi descrito em Souza et al. (2011).

Para o teste de programas concorrentes, inicialmente foram realizados estudos teóricos sobre modelos como redes de Petri, grafos de computação e esquemas de programas paralelos para representação do fluxo de controle de processos concorrentes (Herzog, 1979; Miller, 1973). Em seguida, as técnicas de teste já utilizadas em programas sequenciais começaram a serem adaptadas aos programas concorrentes. Taylor (1983) define o grafo de concorrência para representar as possíveis sincronizações de um programa concorrente.

O modelo apresenta o problema da explosão de estados, pois representa todos estados possíveis do programa. No trabalho posterior (Taylor et al., 1992) apresenta formas de otimizar o grafo de concorrência para um dado programa que se deseja representar. São definidos critérios de cobertura aplicáveis a programas CSP ou ADA, análogos aos critérios definidos para programas sequenciais. Os critérios são definidos seguindo uma hierarquia, dentre os quais destacam-se: todos-caminhos-concorrentes, todas-arestas entre os estados de sincronização, todos-estados, todos-possíveis-rendezvous. Alguns critérios definidos têm sua aplicação limitada pelo problema da explosão de estados.

Yang e Chung (1992) apresentam um modelo de caminhos para representar o comportamento de programas concorrentes, em que são definidos dois grafos: o grafo de fluxo, que modela a visão estática do programa, representando os possíveis fluxos de execução de comandos e o grafo de sincronização, que modela a execução de um programa concorrente, representando as possíveis sincronizações. De acordo com esse modelo, na execução de um programa concorrente, cada processo estará representado no grafo de fluxo formando o conjunto *C-caminho*, e no grafo de sincronização do programa formando o conjunto *C-rota*. Para utilizar a metodologia proposta, são necessárias três questões: Selecionar o *C-caminho*, gerar *C-rotas* e casos de teste e executar os testes. Yang et al. (1998) definiram o Grafo de Fluxo de Programa Paralelo (do inglês PPFG), que representa nós e arestas no contexto de programas paralelos de memória compartilhada. No modelo representado, o PPFG é formado por GFC para cada *thread* (adaptado do GFC de programas sequenciais) e arestas de sincronização entre diferentes *threads*. O critério todos-du-caminhos foi adaptado dos programas sequenciais e foi definido o critério du-pair que relaciona a definição de uma variável v_1 num nó n_i de uma *thread* T_j e um uso da variável v_1 no nó n_i de outra *thread* T_k. O critério é satisfeito se todos os caminhos que compreendem a definição de uma variável e o uso da variável v_1 são exercitados pelo menos uma vez com base no PPFG.

Uma abordagem incremental para teste estrutural foi definida com base na hierarquia entre processos. São construídos grafos de alcançabilidade com base nessa hierarquia, que contêm todas as sincronizações envolvidas entre processos de diferentes níveis. A partir dos grafos são selecionados caminhos capazes de identificar erros de interface envolvendo processos (Koppol e Tai, 1996). Estendendo esse conceito, foi apresentada uma abordagem de teste incremental, consistindo na análise de alcançabilidade para testar seleção de caminhos e teste determinístico para testar execução. Para essa abordagem foi definido um grafo de alcançabilidade chamado *annotated labeled transition system (ALTS)* e com base nisso foram propostos critérios para selecionar caminhos no grafo (Koppol et al., 2002). Uma abordagem de teste que combina três técnicas específicas para análise estática e dinâmica: análise de alcançabilidade, interpretação simbólica e execução controlada é

apresentada em Krawczyk e Wiszniewski (1996). Uma ferramenta de teste (Birman e Renesse, 1994) baseada no sistema Horus foi construída para checar e validar sequências de sincronização e reproduzir uma execução do programa é apresentada em Bechini et al. (1998). Park et al. (2007) apresentada uma ferramenta de detecção *on-the-fly* capaz de detectar e reportar condições de disputa em todos os processos por checar a comunicação entre processos.

Considerando os trabalhos que vem sendo desenvolvidos pelo grupo de Engenharia de Software do ICMC/USP, para programas que usam o ambiente de passagem de mensagem foi definida a ferramenta de teste ValiPVM, que implementa critérios de teste estrutural, suporta geração e avaliação de conjuntos de teste, considerando controle, dados e fluxos de comunicação de programas PVM (Souza et al., 2008a). No contexto de programas em *C* que usam MPI foi definida a ferramenta de teste ValiMPI (Souza et al., 2008b). Para o teste estrutural de programas concorrentes que usam semáforos foi definida a ferramenta de teste ValiPthreads (Sarmanho et al., 2008). Abordagens de teste estrutural para programas concorrentes também foram aplicadas no contexto de web services (Endo et al., 2008), dentre outros trabalhos.

Entre os trabalhos sobre teste de alcançabilidade que consideram programas que usam passagem de mensagem estão Lei e Carver (2006) que definem um modelo para teste de alcançabilidade e propõe a ferramenta RichTest para este teste. Em Wong (2008) são apresentados quatro métodos de geração de sequências de teste a partir de um grafo de alcançabilidade (grafo em que todos os nós representam um estado alcançável e todas arestas representam uma transição entre dois estados alcançáveis). Foi proposto um modelo geral para teste de alcançabilidade que permite que este teste seja aplicado de diferentes maneiras para vários tipos de programas concorrentes, incluindo comunicação síncrona e assíncrona, programas de passagem de mensagem, com memória compartilhada usando semáforos, bloqueios e monitores (Carver e Lei, 2004). Em seguida é proposta uma estratégia para exercitar sequências seletivamente, não exaustivamente, usando uma estratégia de teste combinatorial chamada teste *t-way* (Lei et al., 2007). Estendendo esse trabalho, foi proposta uma estratégia que usa vetor de *timestamps* para determinar a relação do que ocorreu antes entre os eventos recebidos dos pares de sincronização do programa concorrente (Pu e Xu, 2008). Ainda para o teste de alcançabilidade, foi definida uma ferramenta para analisar estatisticamente programas concorrentes (Young et al., 1992).

Sobre abordagens de teste para programas de passagem de mensagem e multithreadeds usando as técnicas de mutação ou perturbação estão operadores de mutação usados para teste de mutação de programas atores (Jagannath et al., 2010). Um conjunto de operadores de mutação para construções concorrentes em SystemC, nessa abordagem foi definida uma métrica para considerar vários escalonamentos que um programa concor-

rente pode gerar para cada entrada (Sen e Abadir, 2010). Um framework (MuTMuT) para redução do teste de mutação de código multithreaded (Gligoric et al., 2010), dentre outros traballhos.

No campo da experimentação sobre critérios de teste para programas concorrentes, em Saini (2005) é descrito um estudo experimental conduzido com o objetivo de avaliar sete critérios de teste para programas concorrentes. Os critérios analisados foram: *All synchronization pair*, *All-branches*, *All decision/condition*, *All du-pair*, *All synchronization-pair and all branches*, *All synchronization-pair and decision/condition* e *All synchronization-pair and all du-pair*. Foi utilizada uma aplicação em Ada para o experimento. Inicialmente foram geradas sequências de sincronização (*SYN-sequences*) (Carver e Tai, 1989) do programa exemplo seguindo cada critério avaliado. Em seguida foi utilizado teste de Mutação para inserção de defeitos e avaliação da eficácia das *SYN-sequences* e, consequentemente, a eficácia dos critérios. Por fim, os resultados para cada critério foram analisados e comparados quanto a eficácia e custo. As seguintes conclusões foram obtidas: o critério *All synchronization pair* foi o menos custoso, porém não foi o mais eficaz. O critério *All branches* foi o mais eficaz que todos os pares de sincronização, porém apresentou um custo maior. O critério *All decision/condition* obteve avaliação mediana entre o menos custoso e o mais eficaz. O critério *All synchronization pair and all du-pair* foi o mais custoso entre os critérios e o mais eficaz. Numa análise mais geral foi possível concluir que o critério *all sync-pair and all branches* apresentou melhor eficácia e custo.

3.4 Considerações Finais

Neste capítulo foram apresentados conceitos sobre o teste de programas concorrentes, enfatizando o modelo GFCP para representação de programas paralelos, critérios de teste e a ferramenta de teste ValiMPI, além de trabalhos relacionados ao tema deste trabalho de mestrado.

Em razão do número reduzido de trabalhos sobre estudos experimentais para avaliação de critérios de teste no contexto de programas concorrentes e da inexistência de um estudo completo sobre os critérios de teste definidos para programas concorrentes que usam MPI, este trabalho trata da realização de um estudo experimental para avaliação destes critérios de teste. No próximo capítulo serão apresentadas as duas primeiras fases do estudo experimental conduzido: a definição e o planejamento.

Capítulo

4

Definição e Planejamento do Experimento Realizado

4.1 Considerações Iniciais

Os conceitos apresentados neste capítulo foram construídos com base no processo experimental definido em Wohlin et al. (2000). Um estudo experimental tem início com a definição, em que o objetivo do experimento é determinado e outras questões são definidas, como o objeto de estudo, o foco de qualidade e o contexto em que o estudo será realizado. Após essa fase, o experimento é planejado com o objetivo de preparar a fase de operação.

Neste capítulo são apresentados a definição e o planejamento do estudo experimental realizado. Na Seção 5.2 são apresentados os principais elementos da fase de definição do experimento. Na Seção 5.3 é apresentado o planejamento do experimento e, na Seção 5.4 são apresentadas as considerações finais deste capítulo.

4.2 Definição do Experimento

Nesta fase foi delineado o objetivo do estudo e forma de condução.

4.2.1 Objetivo

Avaliação da eficácia, custo e *strength* dos critérios de teste definidos para programas concorrentes no contexto de passagem de mensagem que usam o padrão MPI. Os critérios são: todos-nos, todos-nos-r, todos-nos-s, todas-arestas-s, todas-arestas, todos-c-usos, todos-p-usos e todos-s-usos. A avaliação foi realizada para responder as seguintes questões:

- Qual a eficácia em revelar defeitos dos critérios de teste?
- Quais os custos de aplicação dos critérios?
- Qual o *strength* ou dificuldade de satisfação dos critérios?

4.2.2 Metas

- **Objeto de estudo**: critérios de teste todos-nos, todos-nos-r, todos-nos-s, todas-arestas, todas-arestas-s, todos-c-usos, todos-p-usos e todos-s-usos.
- **Propósito**: avaliar o custo, a eficácia e o *strength* dos critérios de teste.
- **Foco da qualidade**: eficácia, custo e *strength*.
- **Perspectiva**: ponto de vista de pesquisadores.
- **Contexto**: este estudo experimental será executado por uma aluna de mestrado sobre um conjunto de programas dos domínios de cálculo, física e bioinformática utilizando uma ferramenta computacional para auxiliar na aplicação dos critérios de teste.

4.3 Planejamento

Nesta fase, o projeto do experimento foi mais detalhado, incluindo também a avaliação dos riscos e ameaças à validade. Foram definidos o contexto, hipóteses, variáveis, participantes, o projeto experimental, a instrumentação e a avaliação de validade do estudo. Após o planejamento o estudo experimental estava totalmente elaborado e pronto para execução.

4.3.1 Seleção do Contexto

O estudo foi realizado pela pesquisadora (mestranda) que o definiu. A ferramenta de teste ValiMPI foi utilizada para execução dos testes. As aplicações para o estudo foram

selecionadas considerando programas já utilizados por outros pesquisadores e uma aplicação real. Inicialmente pretendia-se utilizar um conjunto de programas representativos dos programas concorrentes de memória distribuída, neste caso, a solução seria usar aplicações reais para análise dos critérios de teste. Porém, ao iniciar a busca por programas com código-fonte disponível que pudessem ser utilizados no experimento foi constatado que nem todas as aplicações encontradas poderiam ser utilizadas em razão da dificuldade em encontrar aplicações concorrentes em C que usam o padrão de passagem de mensagem MPI com código fonte disponível e adequado ao experimento pretendido.

Dessa forma, optou-se por utilizar programas que já foram usados por outros pesquisadores em trabalhos de pesquisa e uma aplicação real. O primeiro conjunto consiste de aplicações utilizadas em outras pesquisas (Hausen, 2005; Machado, 2011), que são clássicas da área de programação concorrente, e a aplicação real foi desenvolvida num trabalho de mestrado (Bonetti, 2010) no contexto de algoritmos genéticos. Os programas estão descritos a seguir:

- Máximo Divisor Comum (GCD) que calcula o máximo divisor comum (*greatest common divisor GCD*) entre três números inteiros (Krawczyk e Wiszniewski, 1994).

- Jacobi-Richardson (Jacobi) que calcula o determinante de uma matriz utilizando o método numérico de Jacobi-Richardson.

- Multiplicação de matrizes (Mmult) que multiplica duas matrizes por meio da decomposição do domínio (Quinn, 2003).

- Quicksort (Qsort) que é um algoritmo de divisão-e-conquista que ordena uma sequência de números dividindo-os recursivamente em subsequências menores (Grama et al., 2003).

- Redução (Reduction) que implementa a operação de redução de dados distribuídos (função MPI_Reduce() no MPI), para as operações de adição, multiplicação, maior que e menor que.

- Crivo de Eratóstenes (Sieve) que é uma implementação do algoritmo para encontrar todos os primos menores que um certo número n. O algoritmo utilizado é baseado na solução descrita em Quinn (2003).

- Trap que é uma aplicação para o cálculo de integral.

- Van der Waals que é uma aplicação desenvolvida no contexto de bioinformática para calcular a energia de Van der Waals de uma proteína (Bonetti, 2010).

Com base nos programas e critérios foram gerados conjuntos de teste para os critérios que são 100% adequados a cada critério estudado. Após essa fase já eram conhecidas informações sobre o custo dos critérios. Como as aplicações utilizados no experimento não possuíam defeitos, para avaliar a eficácia dos critérios de teste, foi necessário inserir defeitos nos programas. Durante essa fase foram analisados trabalhos sobre classificação ou taxonomia de defeitos no contexto MPI, porém devido à escassez de trabalhos nessa área, foram pesquisadas taxonomias de defeitos em outros contextos e paradigmas como programas de passagem de mensagem PVM, memória compartilhada e programas tradicionais que pudessem ser mapeados para o estudo pretendido, a fim de possibilitar a obtenção de um conjunto de defeitos representativos dos programas concorrentes. Esses defeitos seriam inseridos nos programas utilizados no experimento para análise da eficácia dos critérios. Após essas pesquisas, optou-se por utilizar a classificação de defeitos proposta em DeSouza et al. (2005) e defeitos descritos em Agrawal et al. (1989). A Tabela 4.1 apresenta os defeitos inseridos. A primeira coluna apresenta cada defeito e a segunda coluna define como o defeito foi inserido no programa.

Tabela 4.1: Defeitos inseridos nos programas do experimento

Número	Descrição do Defeito
1	Defeito no laço que antecede um MPI_Send ou MPI_Recv impedindo que ocorra a mensagem
2	Processo de destino ou origem incorreto num MPI_Send ou MPI_Recv
3	Substituição do processo origem no MPI_Recv por MPI_ANY_SOURCE
4	Acesso a indice de array maior que o limite máximo
5	Uso de variável não inicializada
6	Criação ou uso incorretos de tipos definidos pelo usuário
7	Diferentes tamanhos de mensagem para MPI_Send e MPI_Recv correspondentes
8	Incorreto endereço de um buffer ou de um dado transmitido ou recebido
9	Incorreto número e/ou tipo de parâmentro passado para funções MPI
10	Incorreto tipo de dado em MPI_Send ou MPI_Recv
11	Substituição de mensagens bloqueantes por não bloqueantes (MPI_Isend e MPI_Irecv)
12	Defeito na atribuição a variável usando troca de operador
13	Defeito na atribuição a variável capaz de corromper dados enviados ou recebidos
14	Troca de operador lógico nas estruturas de seleção, repetição ou de controle
15	Retirada de comandos
16	Atribuição de forma incorreta usando variáveis
17	Inserção de pré e pós incrementos ou decrementos nas constantes em MPI_Send e MPI_Recv

Com a execução dos programas com defeitos usando os conjuntos de teste adequados aos critérios, foram obtidas informações sobre a eficácia, possibilitando medir a capacidade dos conjuntos de teste adequados em encontrar os defeitos inseridos.

Após essa fase foram realizadas análises sobre o *strength*. Para essas análises, um conjunto de teste adequado a um determinado critério foi aplicado aos demais critérios com vistas à obtenção da cobertura para aquele conjunto, considerando os demais critérios. Essa atividade foi realizada para cada conjunto de teste, e em seguida as informações obtidas foram cruzadas possibilitando conclusões a cerca do *strength* para os critérios.

4.3.2 Hipóteses

A primeira métrica utilizada para avaliar a efetividade dos critérios foi o custo. Este, pode ser determinado pelo tempo necessário para se construir manualmente os casos de teste, para aprender a utilizar uma ferramenta, ou para aplicação do critério no sentido de analisar associações, caminhos ou mutantes do programa. Neste experimento o custo de aplicação dos critérios de teste estrutural sobre um programa foi medido pela quantidade de casos de teste necessários para a satisfação de um determinado critério, número de elementos requeridos e elementos não executáveis. Com base nessas informações foi definida a seguinte pergunta que pretende ser respondida com este trabalho, considerando para isso uma hipótese nula e uma hipótese alternativa:

Qual o custo dos critérios estruturais para programas concorrentes?

Hipótese Nula(H_0): Não existe diferença de custo entre os critérios de teste estrutural definidos para programas concorrentes.

Hipótese Alternativa(H_1): Ao menos um dos critérios de teste estrutural definidos para programas concorrentes apresenta diferença de custo de aplicação entre os demais critérios.

Admitindo-se que a eficácia de um critério avalia a capacidade em revelar defeitos, pode-se medir a eficácia dos critérios avaliando-se a quantidade de defeitos revelados por um conjunto de casos de teste adequado a um determinado critério em função dos defeitos inseridos que foram revelados para cada programa que comporá o experimento. Com isso, a pergunta 2 foi definida:

Qual a eficácia em revelar defeitos dos critérios de teste estrutural definidos para programas concorrentes?

Hipótese Nula(H_0): Não existe diferença de eficácia entre os critérios de teste definidos para programas concorrentes.

Hipótese Alternativa(H_1): Ao menos um dos critérios de teste estrutural definidos para programas concorrentes apresenta diferença de eficácia de aplicação entre os demais critérios.

Outra avaliação realizada durante o estudo foi com relação ao *strength* dos critérios. Sejam dois critérios de teste C_1 e C_2, admitindo-se que o *strength* de um critério C_1 com relação a outro critério C_2 pode ser medido avaliando-se a cobertura atingida pelo conjunto de teste adequado a C_1 quando aplicado a C_2, foi definida a terceira pergunta:

Qual a relação de inclusão entre os critérios de teste estrutural definidos para programas concorrentes estudados?

Hipótese Nula(H_0): Não existe relação de inclusão entre os critérios de teste estrutural definidos para programas concorrentes.

Hipótese Alternativa(H_1): Ao menos um dos critérios de teste estrutural definidos para programas concorrentes inclui outro critério.

4.3.3 Seleção dos Indivíduos

O experimento foi realizado pela pesquisadora que o definiu, pois envolve conhecimentos de diferentes áreas, como programas concorrentes, teste de software e experimentação, além de conhecimentos sobre os critérios de teste estudados. A pesquisadora cursou disciplinas com o objetivo de abarcar conhecimentos nessas áreas, além da realização de uma revisão sistemática sobre esses conceitos e adquiriu experiências práticas durante o mestrado.

Na inserção de defeitos, para minimizar o viés, a pesquisadora usou classificações de defeitos (definidas em 4.3.1) procurando inserir a maior quantidade de defeitos possíveis nos programas. Os defeitos que não foram inseridos devem-se a particularidades dos códigos dos programas ou limitações da ferramenta de teste, como por exemplo: defeitos relacionados ao uso de operações coletivas, as quais não são tratadas pela ferramenta de teste ValiMPI.

4.3.4 Variáveis

- **Variáveis independentes**:

  1. Critérios de teste;
  2. Programas utilizados;
  3. Conjunto de defeitos inseridos;
  4. Experiência da pesquisadora em aplicar corretamente os critérios.

- **Variáveis dependentes**:

  1. Número de elementos requeridos para cada critério;
  2. Quantidade de casos de teste adequados aos critérios de teste;
  3. Número de elementos não executáveis para cada critério;
  4. Quantidade de defeitos revelados por cada critério;
  5. Porcentagem de cobertura para cada critério considerando um conjunto de teste adequado a um determinado critério;

4.3.5 Descrição da Instrumentação do Experimento

O objetivo do experimento foi a avaliação dos critérios de teste descritos na Seção 4.2.2. Para essa avaliação foi necessário a geração de conjuntos de teste adequados aos critérios que possibilitassem exercitar os programas e analisar o custo, a eficácia e o *strength* do conjunto de teste. O experimento teve início com a construção dos conjuntos de teste que fizeram uso dos programas "isentos" de defeitos, que são GCD, Jacobi, Mmult, Qsort, Reduction, Sieve, Trap e Van der Waals. Após a geração dos conjuntos de teste estavam disponíveis informações sobre o custo de cada critério, considerando que o custo pode ser medido com base na quantidade de casos de teste necessários para satisfação de um determinado critério, número de elementos requeridos e número de elementos não executáveis.

Para avaliação da eficácia, foram inseridos defeitos nos programas, de acordo com a Tabela 4.1. A partir de um programa original foram gerados um determinado número de programas com defeitos dependendo das características do defeito e de cada programa. Como exemplo, considere um programa hipotético que possui apenas uma função de envio de mensagem (MPI_Send()) e uma função de recebimento de mensagem (MPI_Recv()) e o defeito 1 da Tabela 4.1. Um reduzido número de programas com defeitos será gerado devido ao número de funções MPI_Send() e MPI_Recv() e de como elas estão implementadas no código, pois para o defeito 1 considera-se a presença de um laço antecedendo a chamada de função. Caso as chamadas de função não sejam precedidas por um laço não há como inserir esse defeito. Ou caso só exista um laço antecedendo as chamadas, o defeito só pode ser inserido nesse laço de uma ou reduzidas maneiras. Porém, ao considerarmos um programa com várias funções MPI_Send() e MPI_Recv() e vários laços que antecedem essas funções é possível que seja gerado um número maior de programas com defeitos devido à diversidade de locais do código e de formas como o defeito pode ser inserido, considerando os laços que antecedem as chamadas de funções.

Os programas com defeitos foram testados usando os conjuntos de teste adequados gerados a partir de cada programa original para avaliar quão eficientes são os conjuntos de

teste em revelar os defeitos inseridos. Para avaliação da eficácia foi utilizada a quantidade de defeitos revelados pelo conjunto de casos de teste T_1 adequado a um critério C_1 em função dos defeitos inseridos nos programas.

Com a geração dos conjuntos de casos de teste adequados foi possível obter o *strength*, este foi medido utilizando-se um conjunto de casos de teste T_1, adequado a um critério C_1, quando aplicado aos outros critérios C_2, C_3, C_4, C_5, C_6, C_7 e C_8. A cobertura obtida com a aplicação de T_1 a cada critério de teste foi comparada com a cobertura obtida para C_1. Para essa análise, se a cobertura foi baixa é alto o *strength* do critério avaliado em relação a C_1, se pelo contrário a cobertura foi alta, significa que o *strength* é baixo.

4.3.6 Validade

Um fator que pode comprometer a validade interna deste experimento é o grau de experiência da pesquisadora para construção dos casos de teste e inserção dos defeitos, porém, como os casos de teste utilizados são adequados aos critérios, esse fato minimiza a ameaça sobre a experiência da pesquisa na geração dos conjuntos de casos de teste. Para mitigar a ameaça sobre a inserção de defeitos, esta foi realizada seguindo classificações de defeitos conhecidas. Outro fator que pode afetar a validade interna é o domínio e a complexidade dos programas utilizados. Para mitigar esse fator foram utilizados programas de domínios e complexidades variadas desenvolvidos por terceiros. Os programas utilizados não foram desenvolvidos pela pesquisadora que conduziu o experimento para não haver risco de que os programas fossem criados para favorecer os resultados sobre os critérios de teste.

Considerando a validade de construção, uma ameaça poderia ser a satisfação de alguma hipótese ou de um determinado critério favorecidos pela pesquisadora. Neste caso, a utilização da ferramenta de teste ValiMPI atua como fator para mitigação dessa ameaça, quando a ValiMPI gera automaticamente medidas sobre os critérios de teste. Outra ameaça à validade sobre os resultados da eficácia é a utilização de casos de teste gerados a partir dos programas originais (sem defeitos) para testar programas em que são inseridos defeitos. Neste caso alguma característica do programa modificado pode deixar de ser testada. Para essa ameaça não foi encontrado tratamento pois deseja-se verificar a eficácia dos conjuntos de teste gerados a partir dos critérios e dessa forma é necessário que se teste programas com alguma forma de defeitos.

Como aos domínios dos programas utilizados no experimento estão diversificados, esse fator já serve como mitigação as ameaças da validade externa. Outra ameaça foi a existência de apenas um sujeito no experimento, que foi a aluna de mestrado que conduziu o experimento, pois caso houvessem outros sujeitos para inserção de defeitos os resultados poderiam ter sido diferentes. Além disso, o experimento foi realizado em ambiente controlado e não ambiente de produção real, embora os programas tenham sido desenvol-

vidos por terceiros e utilizados pela pesquisadora. Esses motivos tornam a validade desse quasi-experimento comprometida e os resultados e conclusões não podem ser generalizados.

Neste estudo as métricas buscam avaliar a efetividade dos critérios de teste e foram construídas seguindo-se três fatores básicos: eficácia, custo e *strength*. A avaliação estatística foi feita usando métodos adequados, verificando sempre a normalidade do conjunto de dados. Foi aplicado o teste ANOVA *analysis of variance* para conjuntos com distribuição de dados normal (Wohlin et al., 2000) e o teste de Tukey de comparações múltiplas para análise dos grupos e quando a distribuição não era normal foi aplicado o teste não paramétrico Kruskal-Wallis, todos com nível de significância (α) = 0,05.

4.4 Considerações Finais

Neste capítulo foram apresentadas a definição e o planejamento do estudo experimental conduzido com base nos conceitos apresentados por Wohlin et al. (2000) e descritos no Capítulo 2. A definição e planejamento do experimento representam os passos que devem ser seguidos para execução e análise dos resultados. No Capítulo 6 são detalhadas as fases de preparação e execução do experimento.

Capítulo

5

Preparação e Execução do Experimento Realizado

5.1 Considerações Iniciais

Após a definição e o planejamento a próxima fase é a operação do experimento como definido em Wohlin et al. (2000). Tanto a preparação do ambiente quanto a execução são etapas que compõem a operação de um experimento. Na preparação são definidos os indivíduos que participarão do experimento e os recursos para a execução do estudo. Na execução os participantes desempenham as tarefas definidas de acordo com os diferentes tratamentos, finalizando com a coleta de dados.

Neste capítulo são descritos como ocorreram a preparação e a execução do experimento conduzido. Na Seção 5.2 são apresentados os principais conceitos sobre a fase de preparação do ambiente do experimento. Na Seção 5.3 são apresentados conceitos sobre a fase de execução e na Seção 5.4 são apresentadas as considerações finais deste capítulo.

5.2 Preparação do Experimento

Antes que o experimento fosse executado algumas tarefas foram realizadas, como a preparação do material e dos softwares a serem utilizados.

5.2.1 Ambiente de Testes

O ambiente de teste para realização do experimento foi construído usando os seguintes artefatos:

- Ferramenta de teste ValiMPI devidamente instalada e configurada;
- Especificação e código fonte dos programas analisados: GCD, Jacobi, Mmult, Qsort, Reduction, Sieve, Trap e Van der Waals;
- Seleção dos defeitos a serem considerados, segundo a classificação de defeitos proposta em DeSouza et al. (2005), operadores de mutação em Agrawal et al. (1989) e definição de como os defeitos deveriam ser inseridos nos programas.

O ambiente construído para a avaliação dos critérios de testes estudados possuiu as seguintes características:

- Sistema operacional Linux, distribuição Ubuntu - versão 10.04, com a versão 4.1.2 do compilador GCC;
- Ferramenta de teste ValiMPI para testar programas concorrentes em *C* que usam o padrão de passagem de mensagem MPI, oriunda do trabalho de Machado (2011).

5.2.2 Descrição dos Programas

Os programas utilizados no experimento estão implementados na linguagem *C* e utilizam o padrão MPI de passagem de mensagens e foram descritos na Seção 4.3.1. Cada programa foi executado usando 4 processos. A Tabela 5.1 apresenta a quantidade em linhas de código (LOC) e a quantidade de pares de funções de troca de mensagens existentes em cada programa (funções MPI_Send()/MPI_Recv()). Essas informações fornecem uma ideia do grau de complexidade dos programas.

Tabela 5.1: Características dos programas utilizados no experimento

Programa	LOC	Nº Send/Recv
Jacobi	691	11/19
Van der Waals	540	4/4
Qsort	480	7/13
Mmult	192	9/15
Reduction	132	1/1
Sieve	113	3/7
GCD	112	5/5
Trap	77	1/1

5.3 Execução do Experimento

Nesta etapa o experimento foi executado com a aplicação do planejamento elaborado nas fases anteriores.

5.3.1 Geração dos Conjuntos de Casos de Teste

Inicialmente foram gerados os conjuntos de casos de teste adequados a cada critério avaliado, baseados na funcionalidade de cada programa e no código fonte. Após geração de cada caso de teste era verificada se foi atingida a cobertura máxima dos elementos requeridos e caso isso não tivesse ocorrido novos casos de teste eram formulados. Os elementos não executáveis para cada critério eram analisados pela pesquisadora para verificar a cobertura máxima para cada critério. Após essa atividade estavam concluídas as informações sobre o custo de cada critério. Nessa fase, para a análise do custo foram considerados o número de elementos requeridos e número de elementos requeridos não executáveis, representados na Tabela 5.2. Os números de casos de teste adequados a cada critério estão representados na Tabela 5.3.

Tabela 5.2: Elementos requeridos e não executáveis

Programa	**Critérios**							
	t-nos	**t-nos-r**	**t-nos-s**	**t-arestas**	**t-arestas-s**	**t-c-usos**	**t-p-usos**	**t-s-usos**
GCD	62/0	7/0	7/0	25/4	14/4	29/0	40/0	34/17
Jacobi	468/11	37/2	23/2	198/19	67/10	535/74	564/137	107/47
Mmult	131/0	27/0	15/0	207/144	189/144	157/0	100/1	252/204
Qsort	656/246	52/17	28/9	319/146	171/83	1252/668	596/300	300/193
Reduction	76/18	4/2	4/1	28/19	12/9	52/22	72/46	24/21
Sieve	86/0	9/0	7/0	60/18	39/18	84/0	78/4	57/34
Trap	17/0	1/0	3/0	8/0	3/0	10/2	4/1	-
Van der Waals	860/297	16/6	16/10	274/146	66/54	1956/805	1316/567	156/143

Tabela 5.3: Tamanho dos conjuntos adequados

Programa	**Critérios**							
	t-nos	**t-nos-r**	**t-nos-s**	**t-arestas**	**t-arestas-s**	**t-c-usos**	**t-p-usos**	**t-s-usos**
GCD	3	2	2	3	2	4	4	5
Jacobi	5	2	2	7	4	6	7	7
Mmult	1	1	1	4	4	2	5	5
Qsort	4	2	1	6	6	5	5	6
Reduction	1	1	1	1	1	1	1	1
Sieve	1	1	1	3	3	3	4	5
Trap	1	1	1	1	1	1	1	-
Van der Waals	2	1	1	2	1	2	2	2

5.3.2 Semeadura de Defeitos

Para a avaliação da eficácia dos critérios, os defeitos apresentados na Tabela 4.1 foram inseridos nos programas, e a quantidade de versões dos programas geradas para cada defeito podem ser visualizadas na Tabela 5.4. Para cada defeito inserido foi gerada uma

versão diferente do programa original. Tomou-se essa decisão para evitar que um defeito interferisse em outro, mascarando-o e para facilitar a análise da eficácia dos critérios.

Tabela 5.4: Total de defeitos inseridos em cada programa

Defeito	**GCD**	**Jacobi**	**Mmult**	**Qsort**	**Reduct**	**Sieve**	**Trap**	**Waals**
Loop incorreto	2	8	0	2	3	1	2	1
Nº processo incorreto	2	4	1	0	0	2	1	0
Uso de MPI_ANY_SOURCE	1	5	0	2	0	1	0	0
Overflow de array	2	4	3	1	0	4	0	0
Variável não inicializada	0	5	4	0	3	1	4	2
Tipos derivados incorretos	0	0	0	0	0	0	0	3
Diferentes tamanhos de mensagem	0	2	4	1	0	2	4	2
Endereço de buffer incorreto	2	4	2	4	4	6	3	4
Tipo de parâmetro incorreto	1	3	3	1	1	1	1	0
Tipo de dado incorreto	1	1	1	1	0	1	1	2
Uso de operações não bloqueantes	5	1	1	2	2	4	1	2
Troca de operador em atribuição	2	8	5	3	2	8	7	5
Erro nos dados enviados ou recebidos	1	5	5	3	2	5	3	4
Troca de operador nas estruturas	2	5	3	5	1	7	2	4
Retirada de comandos	5	8	6	4	2	7	5	5
Variáveis incorretas em atribuições	4	4	4	5	1	7	0	5
Uso de pré e pós incremento e descremento	0	4	2	1	0	3	0	0
Total	**30**	**71**	**44**	**35**	**22**	**60**	**33**	**39**

Os defeitos foram inseridos com o intuito de que a compilação do programa fosse possível. Assim, considerou-se a inserção de defeitos previstos nas classificações usadas, porém defeitos que não impedissem a compilação do programa. A cada defeito inserido o programa foi executado para verificar essa característica, dessa forma os programas descritos na Tabela 5.4 são capazes de serem executados e não apresentam erros sintáticos.

5.3.3 Execução dos Programas

Após concluída a semeadura de defeitos para os programas foram realizados os testes usando a ferramenta ValiMPI e os conjuntos adequados aos critérios. Os resultados obtidos para a eficácia dos critérios estão descritos na Tabela 5.5. Para cada programa é apresentada a porcentagem de defeitos revelados pelos conjuntos adequados em relação ao total de defeitos inseridos. A primeira coluna especifica cada programa utilizado. Nas

colunas seguintes estão os critérios de teste e a porcentagem de defeitos revelados por cada critério respectivamente.

Tabela 5.5: Dados sobre a eficácia de cada critério de teste

Programa	Defeitos revelados(%)							
	tn	**tnr**	**tns**	**ta**	**tas**	**tcu**	**tpu**	**tsu**
GCD	76,7	76,7	76,7	76,6	76,7	80,0	80,0	100,0
Jacobi	90,1	83,1	83,1	100,0	98,6	97,2	95,8	95,8
Mmult	84,1	84,1	84,1	100,0	100,0	100,0	100,0	100,0
Qsort	100,0	94,3	91,4	100,0	100,0	100,0	100,0	100,0
Reduction	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
Sieve	91,7	91,7	91,7	96,7	96,7	98,3	98,3	100,0
Trap	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-
Van der Waals	100,0	87,2	87,2	100,0	87,2	100,0	100,0	100,0

Após a obtenção de dados referentes à eficácia, o *strength* dos critérios foi analisado, com base no cruzamento de dados com a execução dos programas usando os conjuntos adequados aos critérios quando aplicados aos demais critérios. As Tabelas 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12, 5.13 apresentam os dados obtidos para o *strength* considerando os programas GCD, Jacobi, Mmult, Qsort, Reduction, Sieve, Trap e Van der Waals respectivamente. A primeira coluna das tabelas apresentam os conjuntos adequados a cada critério de teste e as colunas seguintes apresentam as coberturas obtidas pelo conjunto em questão para cada um dos critérios avaliados.

Tabela 5.6: *Strength* para o programa GCD

Strength	**t-nos**	**t-nos-r**	**t-nos-s**	**t-arestas**	**t-arestas-s**	**t-c-usos**	**t-p-usos**	**t-s-usos**
C-t-nos	100,0	100,0	100,0	100,0	100,0	86,2	90,0	76,5
C-t-nos-r	96,8	100,0	100,0	95,2	100,0	65,5	65,0	58,8
C-t-nos-s	96,8	100,0	100,0	95,2	100,0	62,5	65,0	58,8
C-t-arestas	100,0	100,0	100,0	100,0	100,0	86,2	90,0	76,5
C-t-arestas-s	100,0	100,0	100,0	100,0	100,0	86,2	90,0	76,5
C-t-c-usos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	76,5
C-t-p-usos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	76,5
C-t-s-usos	100,0	100,0	100,0	100,0	100,0	96,5	97,5	100,0

Tabela 5.7: *Strength* para o programa Jacobi

Strength	**t-nos**	**t-nos-r**	**t-nos-s**	**t-arestas**	**t-arestas-s**	**t-c-usos**	**t-p-usos**	**t-s-usos**
C-t-nos	100,0	100,0	100,0	87,7	73,7	91,3	82,1	76,7
C-t-nos-r	94,8	100,0	100,0	83,6	78,9	86,3	77,3	66,7
C-t-nos-s	94,8	100,0	100,0	83,6	78,9	86,3	77,3	66,7
C-t-arestas	100,00	100,0	100,0	100,0	100,0	99,8	100,0	100,0
C-t-arestas-s	98,5	100,0	100,0	97,2	100,0	97,0	96,5	91,7
C-t-c-usos	96,5	100,0	100,0	94,4	100,0	100,0	96,9	86,7
C-t-p-usos	100,0	100,0	100,0	100,0	100,0	99,8	100,0	100,0
C-t-s-usos	100,0	100,0	100,0	100,0	100,0	99,8	100,0	100,0

5.4 Considerações Finais

Neste capítulo foi descrito todo processo de execução do experimento, desde as fases iniciais, com a aplicação dos critérios de teste para geração dos conjuntos de teste adequados,

Tabela 5.8: *Strength* para o programa Mmult

Strength	t-nos	t-nos-r	t-nos-s	t-arestas	t-arestas-s	t-c-usos	t-p-usos	t-s-usos
C-t-nos	100,0	100,0	100,0	77,8	68,9	94,3	87,9	56,2
C-t-nos-r	100,0	100,0	100,0	77,8	68,9	94,3	87,9	56,2
C-t-nos-s	100,0	100,0	100,0	77,8	68,9	94,3	87,9	56,2
C-t-arestas	100,0	100,0	100,0	100,0	100,0	100,0	91,9	91,6
C-t-arestas-s	100,0	100,0	100,0	100,0	100,0	100,0	91,9	91,6
C-t-c-usos	100,0	100,0	100,0	85,7	80,0	100,0	84,8	79,2
C-t-p-usos	100,0	100,0	100,0	90,5	86,6	100,0	100,0	87,5
C-t-s-usos	100,0	100,0	100,0	100,0	100,0	100,0	96,0	100,0

Tabela 5.9: *Strength* para o programa Qsort

Strength	t-nos	t-nos-r	t-nos-s	t-arestas	t-arestas-s	t-c-usos	t-p-usos	t-s-usos
C-t-nos	100,0	100,0	100,0	89,0	80,7	97,3	98,3	77,6
C-t-nos-r	98,5	100,0	100,0	74,6	57,9	92,1	91,9	52,3
C-t-nos-s	95,4	97,1	100,0	59,5	43,2	85,3	85,1	35,5
C-t-arestas	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
C-t-arestas-s	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
C-t-c-usos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100.0
C-t-p-usos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	90,6
C-t-s-usos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0

Tabela 5.10: *Strength* para o programa Reduction

Strength	t-nos	t-nos-r	t-nos-s	t-arestas	t-arestas-s	t-c-usos	t-p-usos	t-s-usos
C-t-nos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
C-t-nos-r	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
C-t-nos-s	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
C-t-arestas	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
C-t-arestas-s	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
C-t-c-usos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
C-t-p-usos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
C-t-s-usos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0

Tabela 5.11: *Strength* para o programa Sieve

Strength	t-nos	t-nos-r	t-nos-s	t-arestas	t-arestas-s	t-c-usos	t-p-usos	t-s-usos
C-t-nos	100,0	100,0	100,0	71,4	42,9	92,9	86,5	39,1
C-t-nos-r	100,0	100,0	100,0	71,4	42,9	92,9	86,5	39,1
C-t-nos-s	100,0	100,0	100,0	71,4	42,9	92,9	86,5	39,1
C-t-arestas	100,0	100,0	100,0	100,0	100,0	95,2	89,2	91,3
C-t-arestas-s	100,0	100,0	100,0	100,0	100,0	95,2	89,2	91,3
C-t-c-usos	100,0	100,0	100,0	71,4	42,9	100,0	98,6	52,2
C-t-p-usos	100,0	100,0	100,0	85,7	71,4	100,0	100,0	82,6
C-t-s-usos	100,0	100,0	100,0	95,2	90,5	98,8	98,6	100,0

Tabela 5.12: *Strength* para o programa Trap

Strength	t-nos	t-nos-r	t-nos-s	t-arestas	t-arestas-s	t-c-usos	t-p-usos	t-s-usos
C-t-nos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-
C-t-nos-r	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-
C-t-nos-s	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-
C-t-arestas	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-
C-t-arestas-s	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-
C-t-c-usos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-
C-t-p-usos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-
C-t-s-usos	-	-	-	-	-	-	-	-

até a execução dos programas e obtenção dos resultados. Para efeitos de replicação é importante a descrição minuciosa da execução do experimento para facilitar a reprodução

Tabela 5.13: *Strength* para o programa Van der Waals

Strength	**t-nos**	**t-nos-r**	**t-nos-s**	**t-arestas**	**t-arestas-s**	**t-c-usos**	**t-p-usos**	**t-s-usos**
C-t-nos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
C-t-nos-r	99,8	100,0	100,0	98,4	100,0	98,7	98,1	92,3
C-t-nos-s	99,8	100,0	100,0	98,4	100,0	98,7	98,1	92,3
C-t-arestas	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
C-t-arestas-s	99,8	100,0	100,0	98,4	100,0	98,7	98,1	92,3
C-t-c-usos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
C-t-p-usos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
C-t-s-usos	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0

do mesmo. No Capítulo 6 são apresentadas as análises dos dados obtidos e avaliação das hipóteses propostas para este estudo.

Capítulo

6

Análise dos Resultados

6.1 Considerações Iniciais

Neste capítulo são apresentadas as análises dos resultados e avaliação das hipóteses propostas para a pesquisa. Os resultados são analisados obedecendo as etapas de obtenção dos mesmos. A análise é feita inicialmente usando estatística descritiva, seguida da redução do conjunto de dados e por fim, a avaliação das hipóteses do experimento utilizando testes estatísticos.

Este capitulo está organizado da seguinte forma: Na Seção 6.2 são apresentadas análises segundo a estatística descritiva. Na Seção 6.3 é apresentada a redução do conjunto de dados. Na Seção 6.4 são apresentadas as análises das hipóteses com base em testes estatísticos. Na Seção 6.5 é apresentada uma síntese das análises, e por fim, na Seção 6.6 são apresentadas as considerações finais do capítulo.

6.2 Estatística Descritiva

Para a análise da efetividade dos critérios de teste três métricas foram consideradas: custo, eficácia e *strength*. Dessa forma, a primeira métrica a ser analisada foi o custo dos critérios, cujos dados são apresentados nas Tabelas 5.2 e 5.3.

6.2.1 Análise do Custo dos Critérios

Para melhor visualizar o comportamento da amostra, considerando o número de elementos requeridos para cada critério e cada programa, foi gerado o gráfico de barras representado na Figura 6.1.

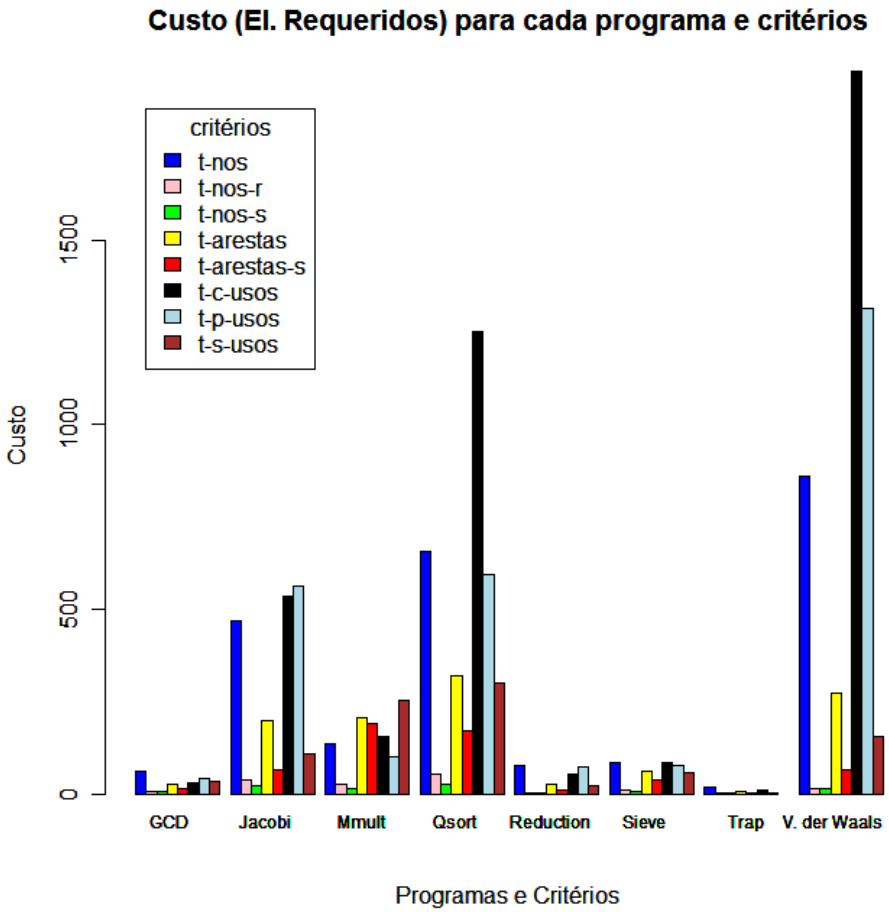

Figura 6.1: Custo para cada critério considerando os programas

Esse gráfico apresenta os programas analisados e seu comportamento quanto ao número de elementos requeridos para cada critério de teste. A análise deste gráfico sugere a existência de grande variabilidade nos dados. Para o programa GCD, representado no primeiro grupo, observa-se que não deve haver variabilidade muito grande na amostra. Porém, ao considerarmos o programa Jacobi o gráfico sugere a existência de muitos elementos requeridos para o critério todos-nos (barra em azul), ao passo que para os critérios todos-nos-r e todos-nos-s, que são subconjuntos do critério todos-nos, o número de elementos requeridos não é tão elevado. O mesmo acontece para o critério todas-arestas que tem uma quantidade de elementos requeridos bem maior que o critério todas-arestas-s que é subconjunto do critério todas-arestas. Para o programa Jacobi, a maior quantidade

de elementos requeridos parece concentrar-se nos critérios todos-c-usos e todos-p-usos. O terceiro grupo representa dados do programa Mmult. Para esse grupo, não há grandes picos, sugerindo a não existência de uma variabilidade muito grande nos dados. O Qsort apresenta comportamento semelhante ao do Jacobi com picos maiores para os critérios todos-c-usos, todos-nos e todos-p-usos, apesar da inversão considerando o GCD para os critérios todos-nos e todos-c-usos, que no GCD o todos-p-usos apresenta mais elementos requeridos que o todos-c-usos. Para os programas Reduction, Sieve e Trap as amostras parecem apresentarem uma variabilidade equilibrada, porém o Van der Waals desponta como o programa que gerou maior variabilidade nos dados. Como representado no Jacobi e Qsort, os critérios todos-c-usos, todos-p-usos e todos-nos são responsáveis pela geração de uma quantidade maior de elementos requeridos. Relacionando os custos dos critérios e analisando os elementos requeridos foi construído outro gráfico de barras, considerando os grupos de critérios, representados na Figura 6.2.

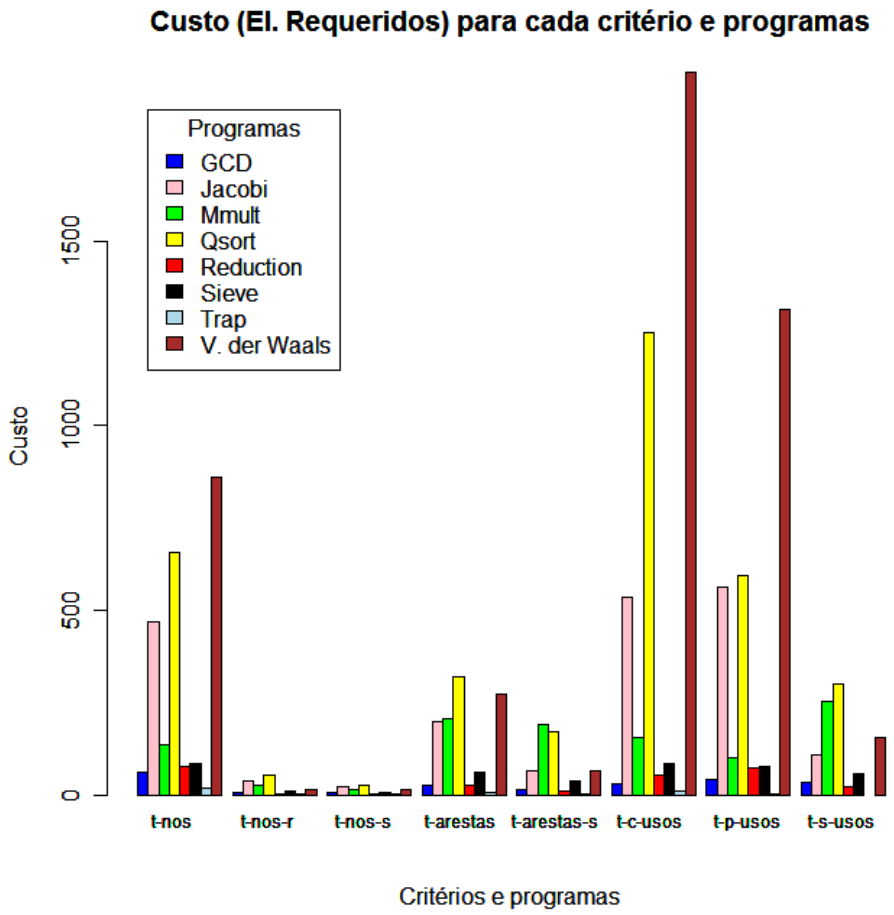

Figura 6.2: Custo para cada programa considerando os elementos requeridos

Com base nesse gráfico observa-se com maior nitidez os critérios que geraram mais e menos elementos requeridos. O critério todos-nos, como observado no gráfico de barras

anterior, gera uma quantidade maior de elementos requeridos para os programas Jacobi (barra rosa), o programa Qsort (barra amarela) e o Van der Waals (barra marrom). Para os critérios todos-nos-r e todos-nos-s a quantidade de elementos requeridos não apresentou uma grande variabilidade. Para os critérios todas-arestas e todas-arestas-s a variabilidade tem um crescimento, porém, com menos destaque do que ocorre com os critérios todos-c-usos e todos-p-usos. O critério todos-s-usos apresenta dados mais próximos que o todos-c-usos e todos-p-usos.

Um resumo da estatística descritiva para cada programa apresentado no gráfico de barras da Figura 6.2 é descrito na Tabela 6.1. Para cada grupo é apresentado o dado mínimo, o dado que representa o primeiro quartil, a mediana, a média, o dado que representa o terceiro quartil e o dado máximo da amostra. Para o primeiro grupo, que representa os dados de custo considerando os elementos requeridos para o critério todos-nos, observa-se que o dado mínimo é 17 e o máximo é 860. Essa grande variabilidade pode ser explicada considerando a mediana da amostra que é 108.5. Como a mediana é uma medida de tendência que divide a amostra ordenada ao meio, pode-se observar que os dados com maior variabilidade encontram-se entre a mediana e o valor máximo. A análise dessa tabela sugere-nos que o critério com menor custo é o todos-nos-s que apresenta como valor mínimo o 3, como máximo 28, mediana 11 e média $12,88$ e como o critério de maior custo o todos-c-usos que apresenta uma grande variabilidade, com mínimo 10, máximo 1956 e mediana $120,5$ confirmando os dados apresentados na Figura 6.2

Tabela 6.1: Análise do custo considerando os elementos requeridos para cada critério

todos-nos	**todos-nos-r**	**todos-nos-s**	**Todas-arestas**
Min. : 17.0	Min. : 1.00	Min. : 3.00	Min. : 8.00
1st Qu.: 72.5	1st Qu.: 6.25	1st Qu.: 6.25	1st Qu.: 27.25
Median :108.5	Median :12.50	Median :11.00	Median :129.00
Mean :294.5	Mean :19.12	Mean :12.88	Mean :139.88
3rd Qu.:515.0	3rd Qu.:29.50	3rd Qu.:17.75	3rd Qu.:223.75
Max. :860.0	Max. :52.00	Max. :28.00	Max. :319.00
todas-arestas-s	**todos-c-usos**	**todos-p-usos**	**todos-s-usos**
Min. : 3.00	Min. : 10.00	Min. : 4.0	Min. : 24.0
1st Qu.: 13.50	1st Qu.: 46.25	1st Qu.: 64.0	1st Qu.: 45.5
Median : 52.50	Median : 120.50	Median : 89.0	Median :107.0
Mean : 70.12	Mean : 509.38	Mean : 346.2	Mean :132.9
3rd Qu.: 93.00	3rd Qu.: 714.25	3rd Qu.: 572.0	3rd Qu.:204.0
Max. :189.00	Max. :1956.00	Max. :1316.0	Max. :300.0
			NA's : 1.0

Outra medida utilizada na análise do custo é a quantidade de elementos não executáveis que está representada no gráfico de barras da Figura 6.3. As informações obtidas com a análise desse gráfico confirmam o que foi observado para os elementos requeridos no gráfico da Figura 6.2. Para os critérios todos-nos-r e todos-nos-s a variabilidade da amostra é pequena, porém para os demais critérios a variabilidade é maior. Para os critérios todos-c-usos e todos-p-usos a variabilidade é maior para os programas Jacobi, Qsort e Van der Waals.

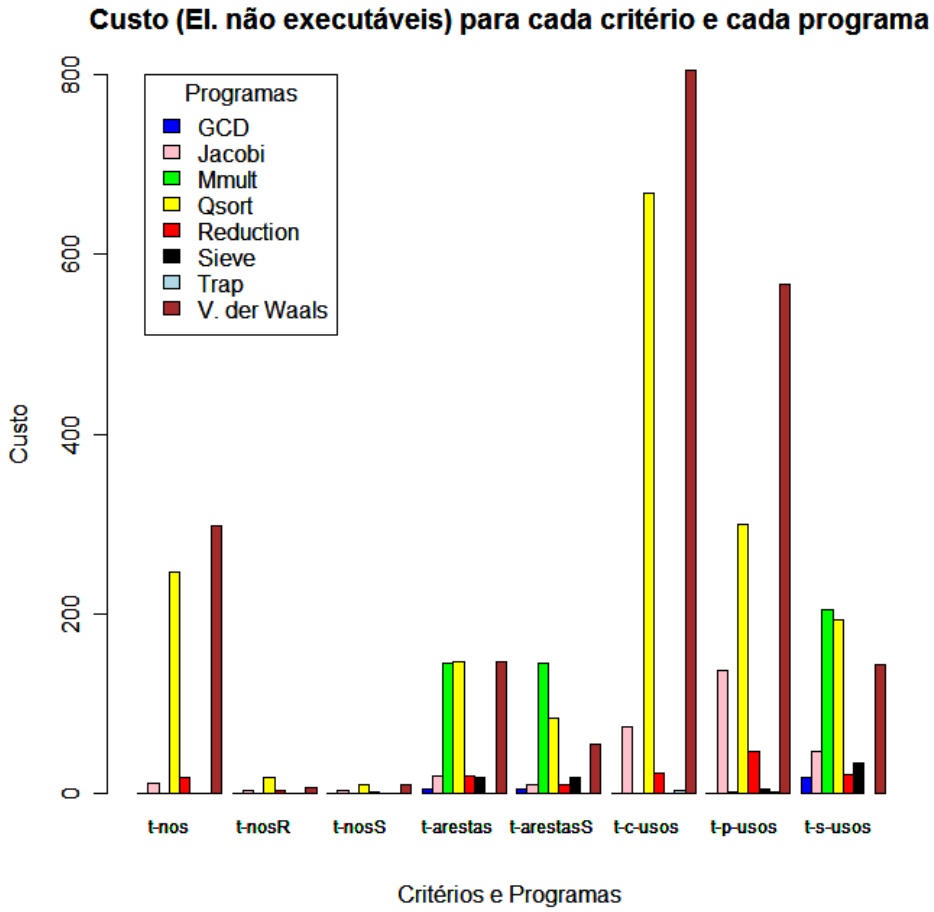

Figura 6.3: Custo para cada critério considerando os elementos não executáveis

Considerando ainda para a análise de custo a quantidade de casos de teste gerados para cada critério, o boxplot representado na Figura 6.4 fornece uma visualização dos dados. Para cada critério de teste foi construído uma caixa no gráfico, com identificação na parte inferior (ta representa o critério todas-arestas, tas o critério todas-arestas-s, tcu representa o critério todos-c-usos, tn foi usado para o critério todos-nos, tnr para o critério todos-nos-r, tns para o critério todos-nos-s, tpu para todos-p-usos e tsu para o critério todos-s-usos).

O gráfico sugere que as maiores variabilidades são obtidas quando considerados os conjuntos de teste adequados aos critérios todas-arestas, todos-p-usos e todos-s-usos. Para a caixa que representa a quantidade de casos de teste para o critério todas-arestas, as quantidades de casos de teste para os programas parecem semelhantes. Para o critério todas-arestas-s a concentração de números de casos de teste ocorre abaixo da mediana para a amostra, que representa o critério todas-arestas, reafirmando a necessidade de uma quantidade menor de casos de teste para o critério todas-arestas. O gráfico sugere que o critério todos-p-usos apresenta maior variabilidade dos dados comparando com o critério

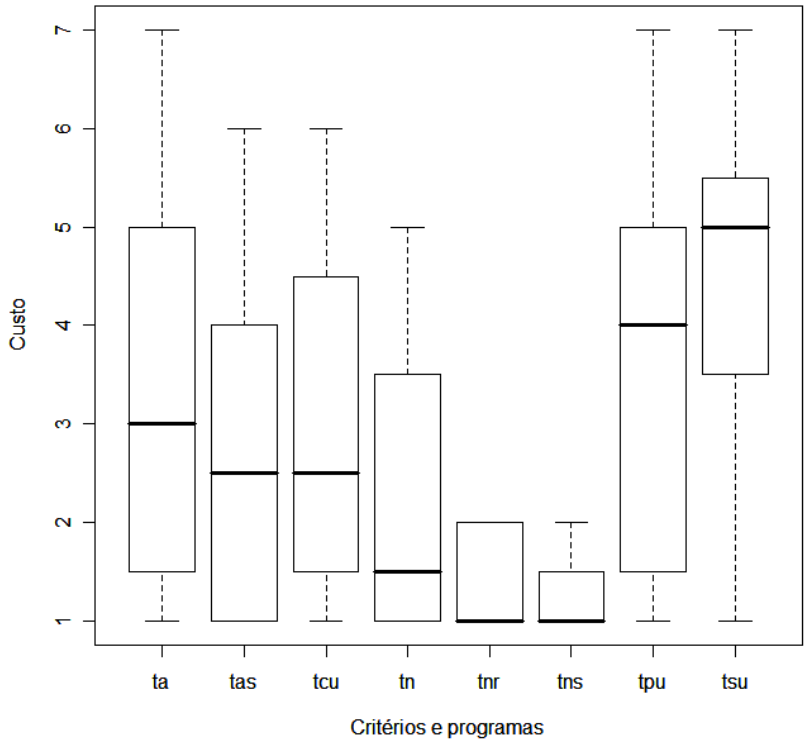

Figura 6.4: Custo para cada critério considerando os casos de teste adequados

todos-s-usos, pois, a parte inferior da caixa para o critério todos-p-usos apresenta-se com mais volume. Isso pode significar que o critério todos-s-usos exige uma quantidade de casos de teste elevada, porém mais estável que o critério todos-p-usos. Observando os critérios todos-nos, todos-nos-r e todos-nos-s, estes apresentam números de casos de teste bem menores que os demais critérios, observando-se tanto a mediana quando os limites de máximo e mínimo de cada amostra.

Dessa forma, a análise descritiva sobre o custo sugere que os critérios baseados no fluxo de dados e em passagem de mensagem apresentam custos mais elevados que os critérios baseados no fluxo de controle e em comunicação, resultado similar aos obtidos para programas sequenciais quando comparado o fluxo de controle com fluxo de dados. O teste de hipóteses fornece maiores indícios sobre essas suposições.

6.2.2 Análise da Eficácia

Para a análise da eficácia, os dados foram representados no boxplot da Figura 6.5. Notam-se medianas acima de 83%, o que representa, no geral, alta eficácia para os critérios. Os critérios baseados no fluxo de controle e em comunicação apresentam medianas mais baixas que os critérios baseados no fluxo de dados e em passagem de mensagem.

O gráfico também apresenta pontos fora dos limites máximo e mínimo de cada caixa do gráfico, que são os *outliers* para essa amostra.

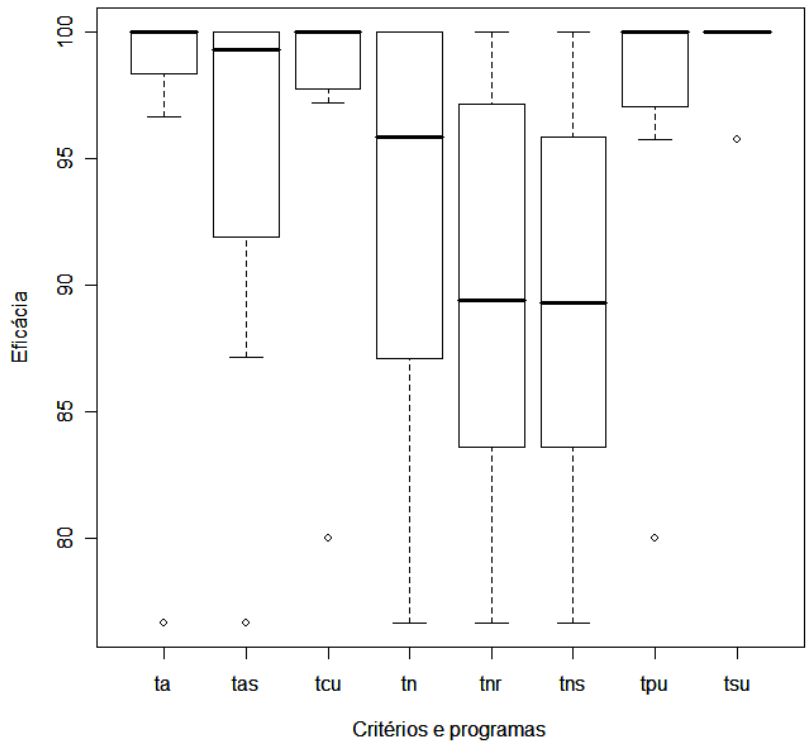

Figura 6.5: Boxplot para a eficácia dos critérios

Para reforçar a análise da eficácia foi construído um gráfico de barras representado na Figura 6.6, que ilustra o comportamento dos dados obtidos. Com a análise desse gráfico, nota-se uma concentração de barras maiores para os critérios baseados no fluxo de dados e em passagem de mensagem, indicando que para esses critérios a eficácia é maior, e de barras de tamanho variável e menores para os critérios de fluxo de controle e comunicação, indicando eficácia menor para esses critérios. Considerando o critério todos-s-usos, apesar do gráfico não apresentar dados para um dos programas, pois, este programa não gera elementos requeridos para esse critério, o critério todos-s-usos possui barras mais estáveis e maiores, podendo representar um indício de que o conjunto de teste adequado a este critério fornece maiores coberturas para os programas comparando-o com os conjuntos adequados ao demais critérios. Observando os critérios de fluxo de controle e comunicação, os critérios todos-nos-r e todos-nos-s assemelham-se em comportamento, sugerindo menor eficácia. O teste de hipóteses avalia essas indicações e as hipóteses definidas anteriormente.

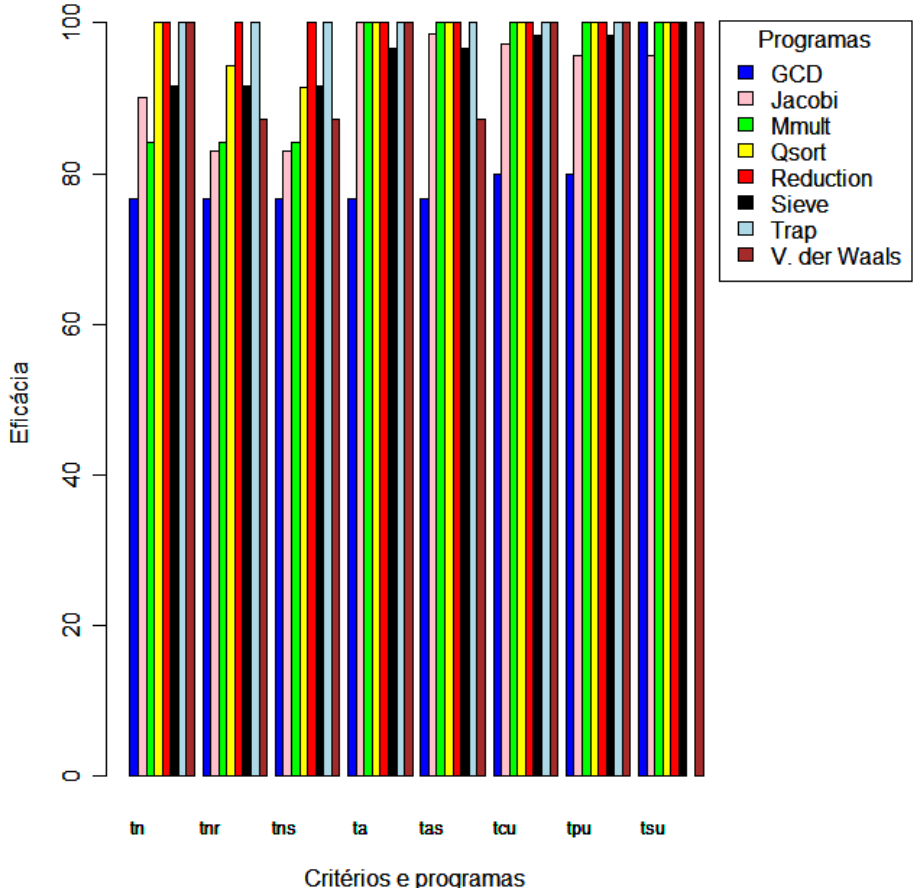

Figura 6.6: Eficácia para os critérios de teste

6.2.3 Análise do Strength

Para o *strength* os dados obtidos foram organizados tendo-se como base os conjuntos de teste adequados a cada critério. A Figura 6.7 apresenta os dados obtidos considerando os conjuntos de teste adequados ao critério todos-nos quando aplicados aos demais critérios. Os três primeiros grupos representados na figura comprovam que os critérios todos-nos-r e todos-nos-s possuem baixo *strength* ou dificuldade de satisfação em relação ao critério todos-nos, fato que pode ser explicado em razão dos critérios todos-nos-r e todos-nos-s serem subconjuntos do critério todos-nos. Analisando os critérios todas-arestas e todas-arestas-s, o conjunto de teste adequado ao critério todos-nos atinge 100% de cobertura para 50% dos programas testados. Para o critério todas-arestas a cobertura média é maior que 90% e para o critério todas-arestas-s a cobertura média é $83,26\%$. Os critérios todos-c-usos e todos-p-usos apresentaram comportamentos semelhantes para os programas testados. Para o critério todos-c-usos a média de cobertura foi 95.2% e para o critério todos-p-usos 93.8%. Os conjuntos adequados ao critério todos-nos quando aplicados ao critério todos-s-usos alcançam menores coberturas. Para dois pro-

gramas a cobertura para esse critério atingiu 100%, porém para os demais programas, a média de cobertura foi 78.2%. Com base nessa análise preliminar, para os conjuntos de teste adequados ao critério todos-nos a maior dificuldade foi encontrada para cobrir o critério todos-s-usos, e em ordem decrescente todas-arestas-s, todas-arestas, todos-p-usos, todos-c-usos, todos-nos-r e todos-nos-s.

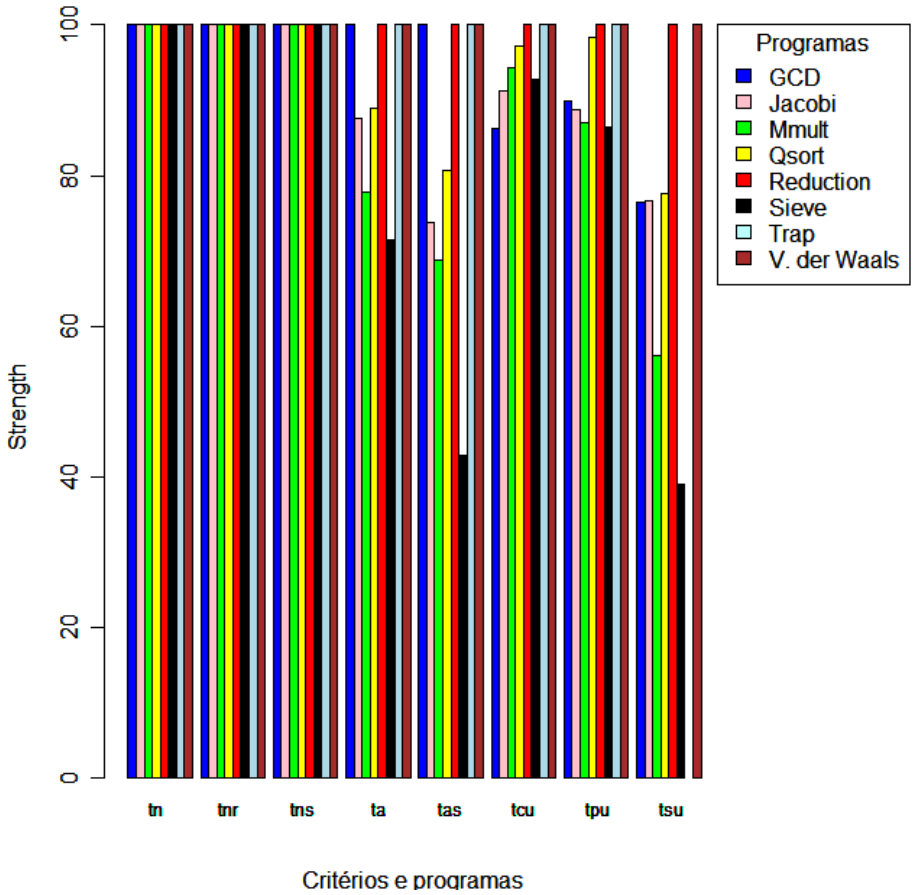

Figura 6.7: *Strength* para o conjunto de teste adequado ao critério todos-nos

A Figura 6.8 apresenta o *strength* ou dificuldade de satisfação para os conjuntos de teste adequados ao critério todos-nos-r quando aplicados aos demais critérios para os programas avaliados. Para os conjuntos adequados ao critério todos-nos-r quando aplicados ao critério todos-nos observa-se que a cobertura foi alta, e na média atingiu 98.7%. Para o critério todos-nos-s a cobertura atingiu 100% para todos os programas avaliados. Para o critério todas-arestas em geral as coberturas são altas, acima de 70%, na média foi 87.5%. Para o critério todas-arestas-s não há um padrão de cobertura para os programas, o que pode ser consequência das diferenças de complexidade entre eles, a média de cobertura foi 81%. Para o critério todos-c-usos as coberturas comportaram-se de formas semelhantes para a maioria dos programas, com uma média de 90.6%. Para o critério todos-p-usos a média

já não foi tão alta, quanto para o critério todos-c-usos, atingindo 88.3%. Como para o conjunto de teste todos-nos, os conjuntos adequados a todos-nos-r obtiveram cobertura menor para o critério todos-s-usos (66.5% em média). Dessa forma, o *strength* para os conjuntos de teste adequados ao critério todos-nos-r em ordem decrescente, com base nessa análise foi: todos-s-usos, todas-arestas-s, todas-arestas, todos-p-usos, todos-c-usos, todos-nos e todos-nos-s.

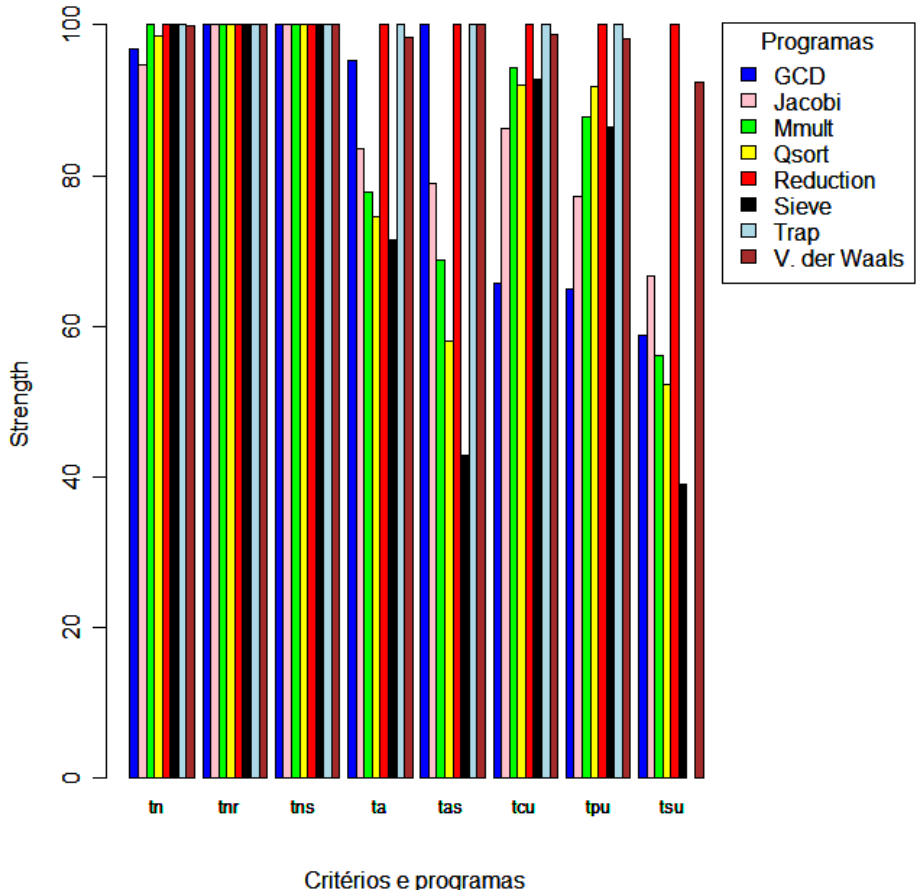

Figura 6.8: *Strength* para o conjunto de teste adequado ao critério todos-nos-r

A Figura 6.9 apresenta os dados obtidos para os conjuntos adequados ao critério todos-nos-s quando aplicados aos demais critérios. O comportamento desses conjuntos de teste assemelharam-se ao dos conjuntos todos-nos-r discutidos na análise anterior. Para o critério todos-nos a cobertura atingiu 98.3%, para o critério todos-nos-r a média foi 99.6%, para o critério todas-arestas 85.7% e para todas-arestas-s 79.2%. Para os critérios todos-c-usos, todos-p-usos e todos-s-usos 89.9%, 87.1%, 64.1% respectivamente. Com base nessa análise, em ordem decrescente de dificuldade de satisfação para os conjuntos todos-nos-s estão: todos-s-usos, todas-arestas-s, todas-arestas, todos-p-usos, todos-c-usos, todos-nos e todos-nos-r.

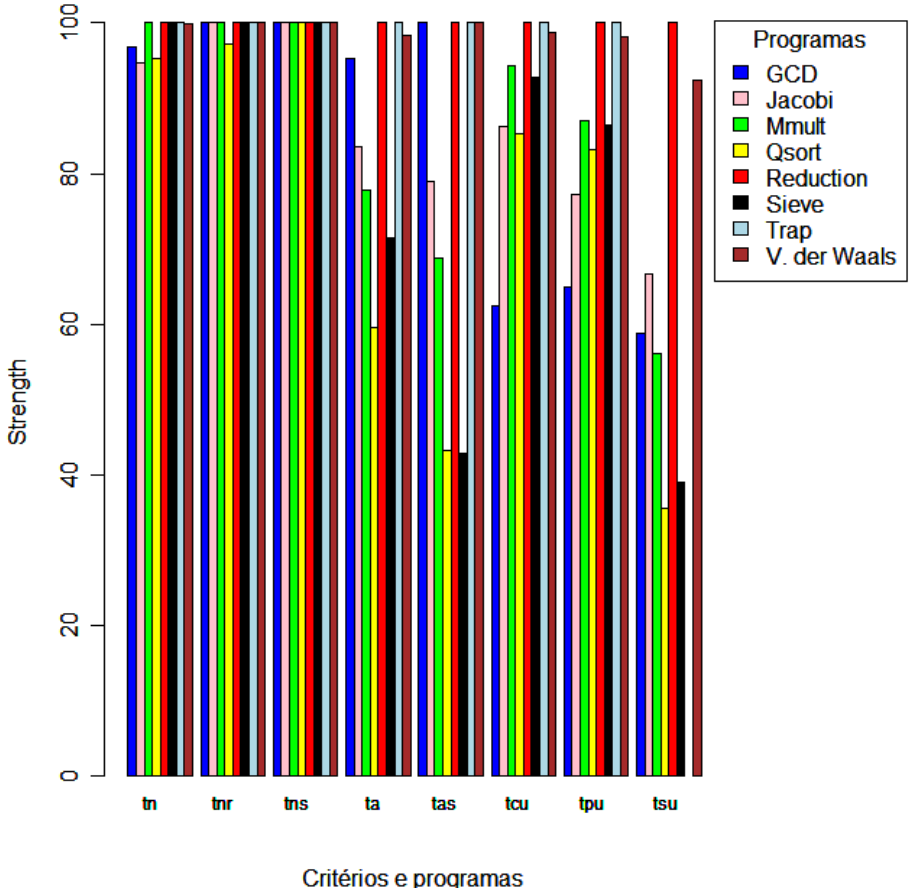

Figura 6.9: *Strength* para o conjunto de teste adequado ao critério todos-nos-s

Para os conjuntos de teste adequados ao critério todas-arestas, cujos dados podem ser visualizados na Figura 6.10, o *strength* apresentou-se alto para os critérios todos-nos, todos-nos-r, todos-nos-s e todas-arestas-s (100%). Para o critério todos-c-usos apenas três programas não apresentaram cobertura máxima que foram o GCD, o Jacobi e o Sieve com coberturas médias $86,2\%$, 99.78% e 95.24%, fornecendo uma cobertura média para esse critério de $97,65\%$. Para o critério todos-p-usos 3 programas não atingiram cobertura máxima, gerando uma média de cobertura para esse critério de $96,4\%$. Para o critério todos-s-usos a cobertura média foi 94.2%. Assumindo os dados sugeridos pelo gráfico, o *strength* em ordem decrescente pode ser exposto da seguinte forma: todos-s-usos, todos-p-usos, todos-c-usos, todos-nos, todos-nos-r, todos-nos-s, todas-arestas-s.

A Figura 6.11 fornece dados da aplicação dos conjuntos adequados ao critério todas-arestas-s aos demais critérios. Nota-se que a cobertura para os critérios de fluxo de controle e fluxo de comunicação são altas. Para o critério todos-nos a cobertura média foi $99,8\%$. Para os critérios todos-nos-r e todos-nos-s a cobertura foi 100%. Para o critério todas-arestas a cobertura média atingiu 99.4%. O critério todos-c-usos obteve como cobertura média

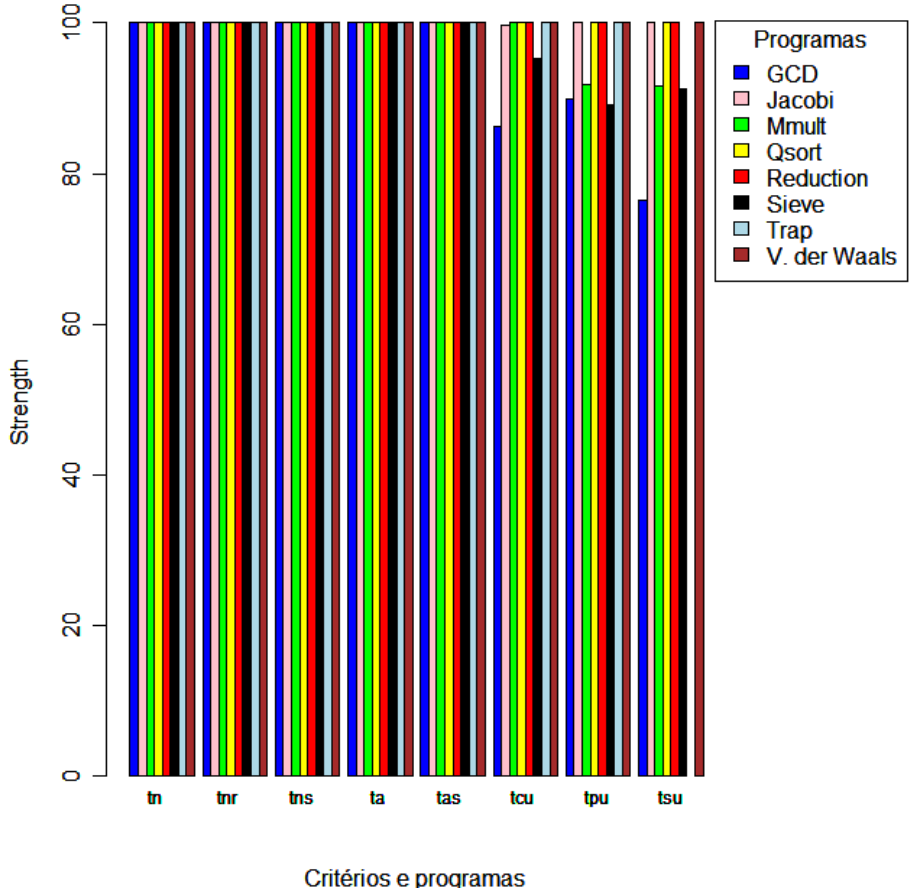

Figura 6.10: *Strength* para o conjunto de teste adequado ao critério todas-arestas

91.2%. Para o critério todos-p-usos e todos-s-usos as coberturas médias foram 88.3% e 66.5%. O *strength* para esses conjuntos de teste em ordem decrescente foi: todos-s-usos, todos-p-usos, todos-c-usos, todas-arestas, todos-nos, todos-nos-r e todos-nos-s.

A Figura 6.12 apresenta o conjunto de dados gerado com a aplicação dos conjuntos de teste adequados ao critério todos-c-usos em relação aos demais critérios. Para o critério todos-nos apenas o programa Jacobi não obteve cobertura 100%, de forma que para esse critério a média foi 99.6%. Para os critérios todos-nos-r e todos-nos-s a cobertura foi 100%. Porém, para o critério todas-arestas a cobertura média foi 93.9%. Para o critério todas-arestas-s 90.3%. Para o critério todos-p-usos a cobertura média obteve 97.5%. Para o critério todos-s-usos $84,9\%$. Concluindo, o gráfico sugere que o *strength* para os conjuntos adequados ao critério todos-c-usos em ordem decrescente é: todos-s-usos, todas-arestas-s, todas-arestas, todos-p-usos, todos-nos, todos-nos-r e todos-nos-s.

Para representar o comportamento dos conjuntos adequados ao critério todos-p-usos foi gerado o gráfico ilustrado na Figura 6.13. Com a observação desse gráfico, é possível notar uma tendência para os critérios todos-nos, todos-nos-r e todos-nos-s apresentarem

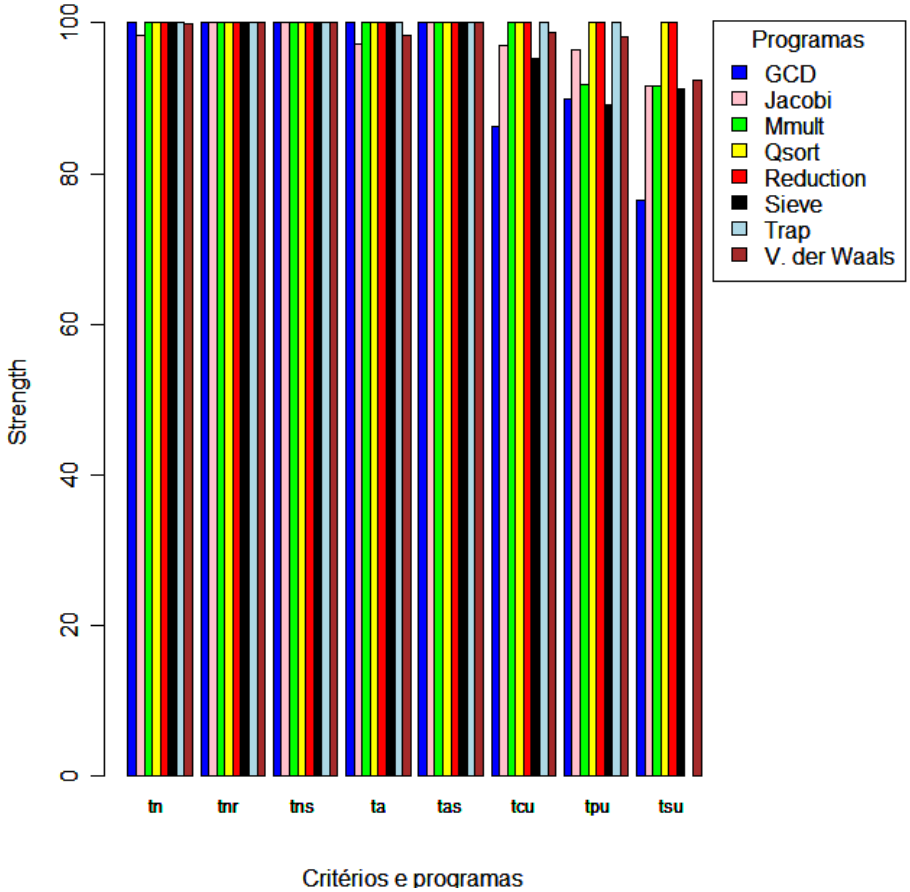

Figura 6.11: *Strength* para o conjunto de teste adequado ao critério todas-arestas-s

coberturas bem altas (100% para os três casos). Para os demais critérios as coberturas não chegam ao máximo, mas continuam altas. Para o critério todas-arestas a média foi 97%, para o critério todas-arestas-s 94.7%. Para o critério todos-c-usos a cobertura média foi muito próxima de 100% e para o critério todos-s-usos a cobertura média foi 91%. Dessa forma, o *strength* para o conjunto de teste avaliado, em ordem decrescente é: todos-s-usos, todas-arestas-s, todas-arestas, todos-c-usos, todos-nos, todos-nos-r e todos-nos-s.

O gráfico que representa o *strength* para os conjuntos de teste adequados ao critério todos-s-usos pode ser visualizado na Figura 6.14. Notam-se coberturas mais altas para todos os critérios considerando este conjunto de teste. Os critérios todos-nos, todos-nos-r e todos-nos-s atingem coberturas 100%. Para o critério todas-arestas apenas o programa Sieve não obteve cobertura máxima. Para o critério todas-arestas-s o mesmo ocorreu. Para o critério todos-c-usos houve pequenas flutuações em torno da cobertura máxima(99.3% em média). Para o critério todos-p-usos a cobertura média foi de 98.9%. Considerando a ordem decrescente para o *strength* do conjunto todos-s-usos, tem-se: todos-p-usos, todos-c-usos, todas-arestas-s, todas-arestas, todos-nos, todos-nos-r e todos-nos-s.

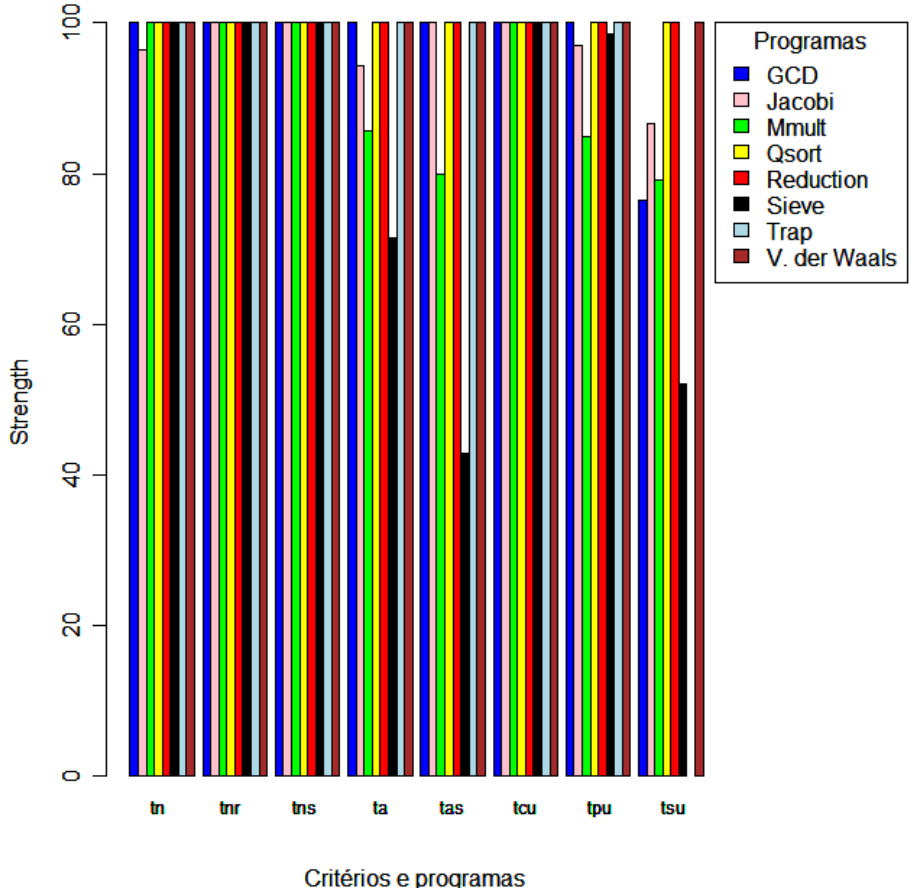

Figura 6.12: *Strength* para o conjunto de teste adequado ao critério todos-c-usos

6.3 Redução do Conjunto de Dados

Dois dos programas utilizados no experimento (Reduction e Trap) são simples tanto em quantidade de linhas de código quanto no número de funções MPI, como pode ser observado na Tabela 5.1. Essas características geraram medidas máximas de cobertura, e portanto, de eficácia e *strength* para esses programas considerando qualquer critério de teste. Como consequência, os resultados obtidos para o grupo podem representar medidas mais elevadas do que as reais, inserindo viés nos resultados para a eficácia e *strength*.

Tabela 6.2: Resultados da ANOVA para o custo com base no número de elementos requeridos para cada critério

	Df	**Sum Sq**	**Mean Sq**	**F value**	**Pr(>F)**
Critérios	7	66.81	9.54	4.94	0.0002213
Residuals	55	106.26	1.93		

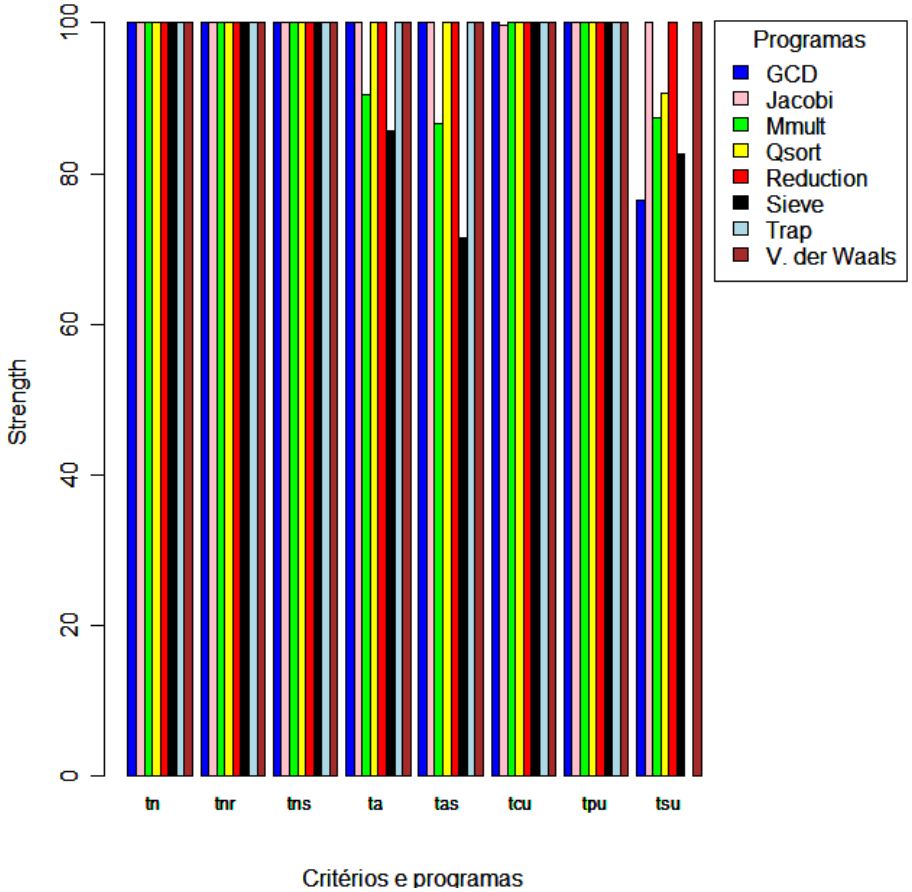

Figura 6.13: *Strength* para o conjunto de teste adequado ao critério todos-p-usos

Para mitigar essa ameaça, optou-se por excluir os dados desses programas no teste das hipóteses relacionadas à eficácia e ao *strength*.

6.4 Teste de Hipóteses

6.4.1 Análise do Custo

Para avaliação da primeira hipótese, o custo foi verificado com base na quantidade de elementos requeridos, em seguida, com base na quantidade de elementos não executáveis e por fim, pela quantidade de casos de teste adequados a cada critério. O primeiro teste realizado considerou a quantidade de elementos requeridos para cada critério de teste. Neste caso, como foi possível verificar que a suposição de normalidade dos erros não estava sendo violada, por meio do teste de normalidade de Shapiro-Wilk, cujo *p-value* para a amostra foi $>$ que $0,05$, significando normalidade dos erros, foi realizada uma

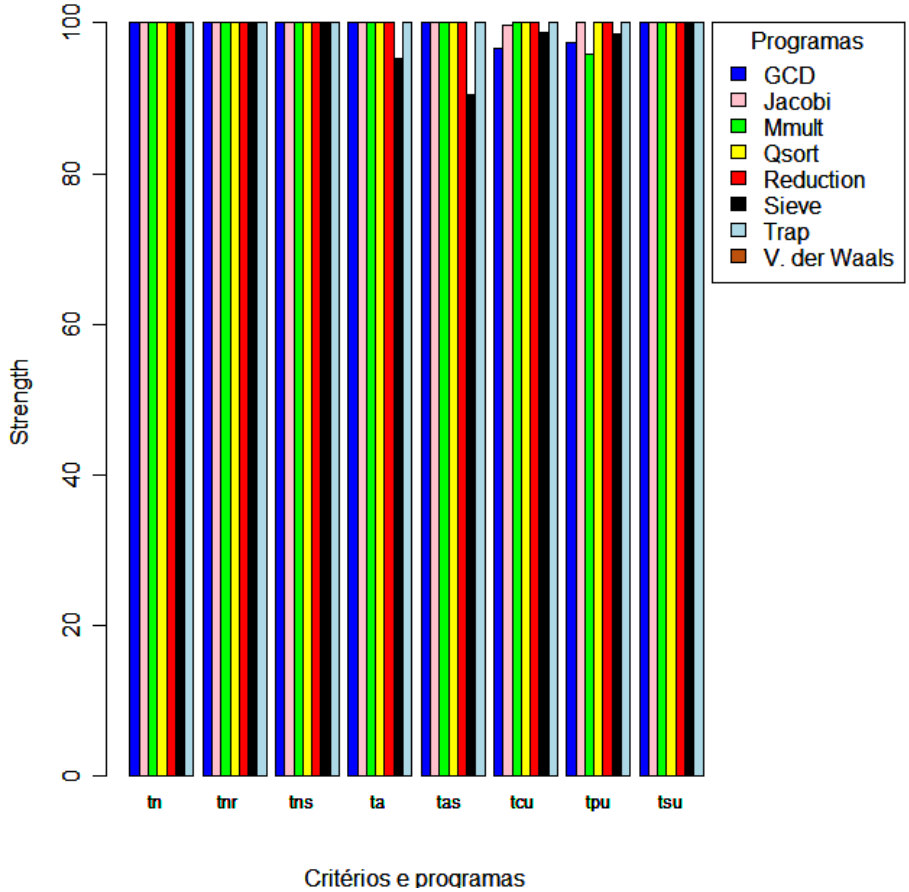

Figura 6.14: *Strength* para o conjunto de teste adequado ao critério todos-s-usos

análise paramétrica, por meio do modelo ANOVA (*analysis of variance*)(Wohlin et al., 2000), cujos resultados são apresentados na Tabela 6.2.

Na primeira coluna da tabela estão os graus de liberdade para a amostra, na segunda coluna a soma dos dados da amostra, na terceira coluna a média, na quarta coluna o valor do *F-value* determinado pela ANOVA e na quinta coluna o *p-value*. Para os testes relacionados à ANOVA foi assumido o nível de significância de 5%. Neste caso, como o *p-value* obtido foi 0.0002 e portanto, $<$ que 0.05 deve-se rejeitar a hipótese nula H_0. O teste sugere que existe diferença de custo para ao menos um dos critérios de teste estrutural definidos para programas concorrentes, como afirmado na hipótese alternativa H_1. De posse dessa informação, foi aplicado o teste de Tukey para comparações múltiplas de médias com o objetivo de relacionar os dados referentes aos critérios de teste e selecioná-los par a par. A Figura 6.15 apresenta essa análise.

Rejeita-se a hipótese de igualdade entre todos os pares de critérios para *p-values* $<$ ou $=$ 0.05 (nível de significância). Os pares de critérios para os quais a diferença entre as médias é significativa são identificados pelos intervalos de confiança (os que não contém o valor 0).

Observando a figura, para 6 pares de critérios há indícios de diferenças significantes. Os pares de critérios são: todos-nos-r e todos-c-usos, todos-nos-s e todos-c-usos, todos-nos-r e todos-p-usos, todos-nos-s e todos-p-usos, todos-nos-r e todos-nos e todos-nos-s e todos-nos. Esses relacionamentos entre os critérios, quando analisados juntos aos dados representados no boxplot da Figura 6.16 sugerem que os critérios todos-nos, todos-c-usos e todos-p-usos estão entre os mais custosos e os critérios todos-nos-r e todos-nos-s os menos custosos. O boxplot ilustra as diferenças entre os custos para esses critérios com base nas medianas e na variabilidade da amostra.

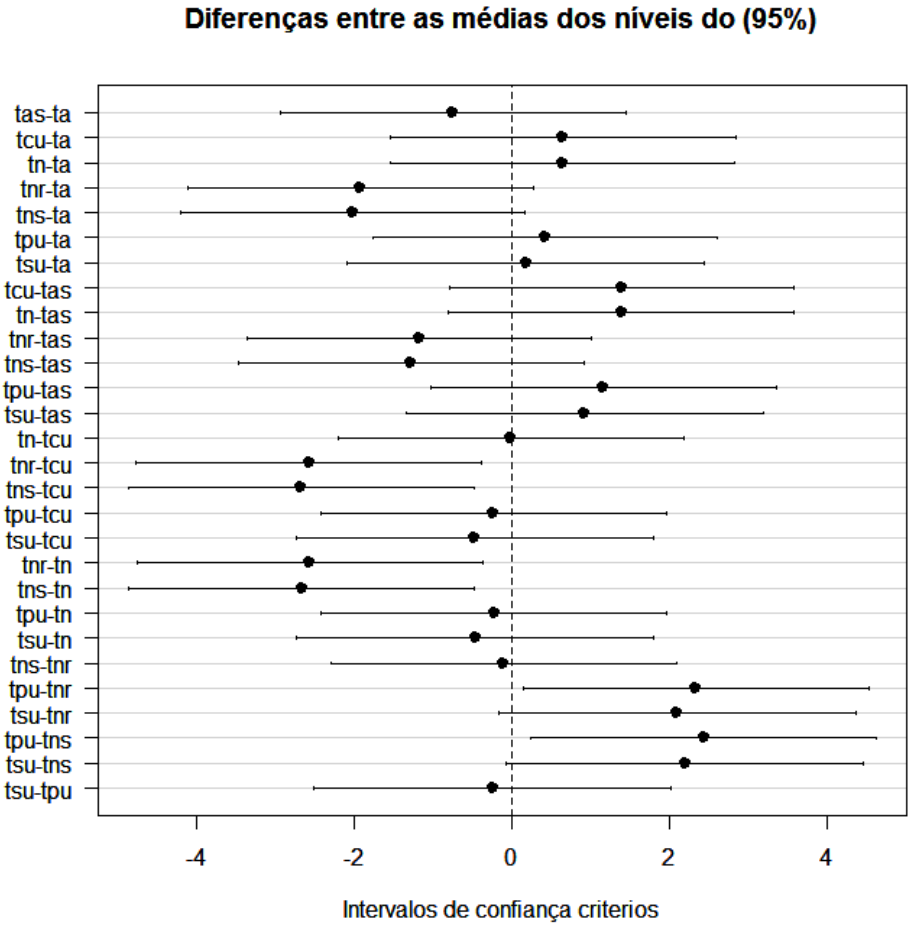

Figura 6.15: Resultados do teste de Tukey para o custo dos critérios com base no nº de elementos requeridos

O custo também foi analisado com base na quantidade de elementos não executáveis. Verificou-se, também neste caso, por meio do teste de normalidade de Shapiro-Wilk, que a suposição de normalidade dos erros não estava sendo violada, desta forma, o modelo ANOVA foi novamente utilizado. Os resultados estão ilustrados na Tabela 6.3. Com a observação do *p-value*< 0.05, deve-se rejeitar a hipótese nula H_0 e aceitar a hipótese alternativa H_1 que diz que ao menos um critério de teste apresenta custo distinto dos

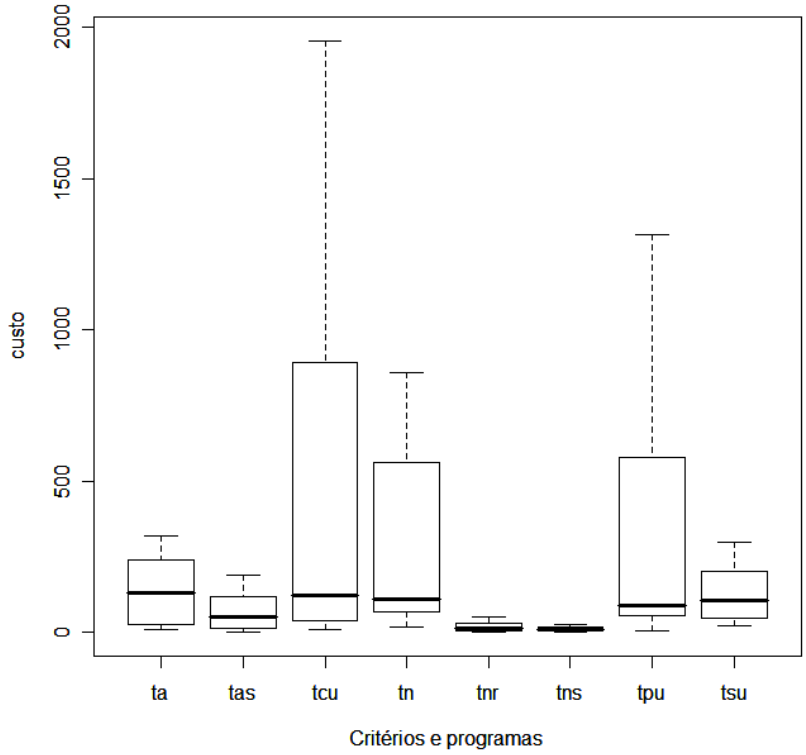

Figura 6.16: Boxplot para o custo dos critérios com base no número de elementos requeridos

demais. Em virtude disso, foi aplicado o teste de Tukey de comparações múltiplas para verificar para quais critérios há diferenças significativas. Os resultados para esse teste estão sumarizados na Figura 6.17, em que é possível visualizar a existência de diferenças significativas entre os pares de critérios: todos-s-usos e todos-nos-r e todos-s-usos e todos-nos-s. Esse resultado, quando analisado em consonância com o gráfico de barras da Figura 6.3 possibilita identificar como critério mais custoso o todos-s-usos e como os menos custosos os critérios todos-nos-r e todos-nos-s.

Tabela 6.3: Resultados da ANOVA para o custo com base no número de elementos não executáveis

	Df	**Sum Sq**	**Mean Sq**	**F value**	**Pr(>F)**
Critérios	7	42.115	6.016	2.8202	0.01388
Residuals	55	117.332	2.133		

O teste não paramétrico Kruskal-Wallis para comparações múltiplas foi utilizado para a análise do custo com base na quantidade de casos de teste adequados, em razão do teste de normalidade de Shapiro-Wilk ter detectado violação na normalidade do erros. Este teste

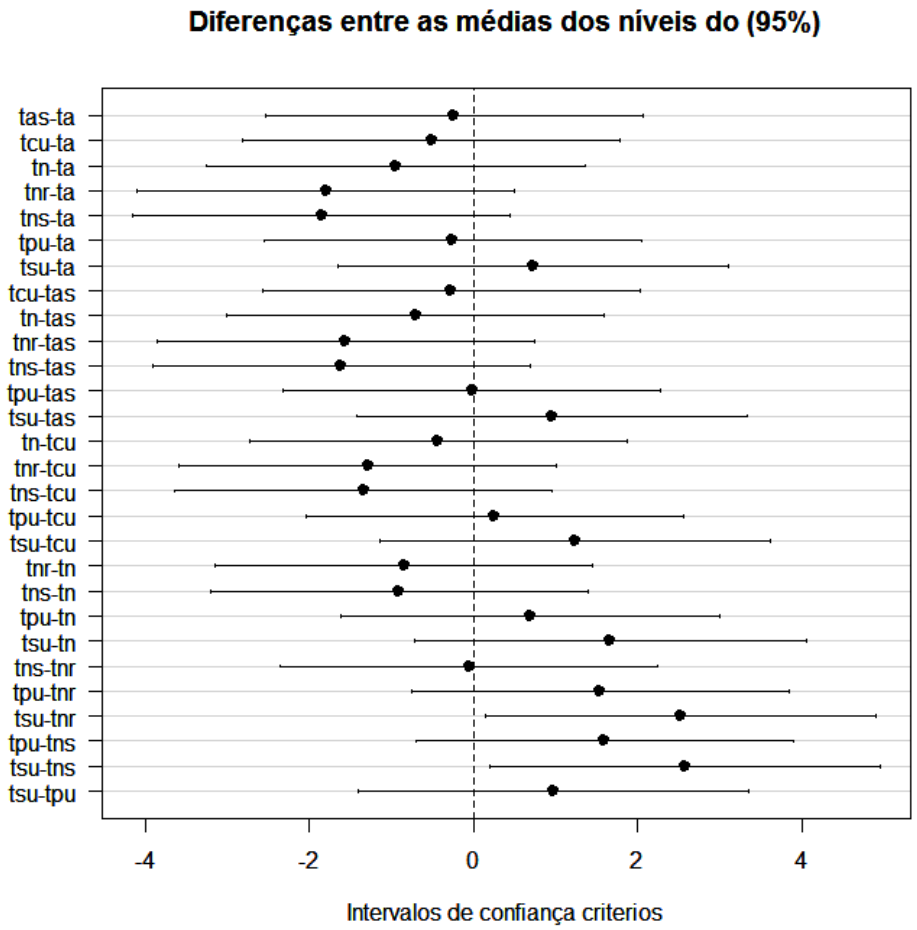

Figura 6.17: Resultados do teste de Tukey para o custo com base no nº de elementos requeridos não executáveis

estatístico utiliza uma análise par a par de cada critério verificando diferenças significativas entre eles. Os resultados do teste podem ser visualizados na Figura 6.18. A primeira coluna da tabela especifica a os pares de critérios comparados, a segunda coluna apresenta a diferença observada para aquele par de critérios, a terceira coluna sumariza a diferença crítica para o par de critérios e a quarta coluna apresenta o resultado do teste considerando a existência ou ausência de diferença significativa para cada par de critérios. O *true* indica que há diferença e *false* indica que o teste não identificou diferença significativa para aquele par de critérios. Observa-se que nenhuma das combinações apresentam diferença significativa, pois para todos os valores observados o teste aponta ausência de diferença significativa. Neste caso, nada pode ser afirmado sobre a existência de diferença de custo dos critérios com base no número de casos de teste adequados.

```
Multiple comparison test after Kruskal-Wallis
p.value: 0.05
Comparisons
          obs.dif critical.dif difference
acu-ae   2.187500     28.62950      FALSE
acu-aes  3.187500     28.62950      FALSE
acu-an   8.000000     28.62950      FALSE
acu-anr 15.875000     28.62950      FALSE
acu-ans 18.187500     28.62950      FALSE
acu-apu  3.875000     28.62950      FALSE
acu-asu 10.473214     29.63435      FALSE
ae-aes   5.375000     28.62950      FALSE
ae-an   10.187500     28.62950      FALSE
ae-anr  18.062500     28.62950      FALSE
ae-ans  20.375000     28.62950      FALSE
ae-apu   1.687500     28.62950      FALSE
ae-asu   8.285714     29.63435      FALSE
aes-an   4.812500     28.62950      FALSE
aes-anr 12.687500     28.62950      FALSE
aes-ans 15.000000     28.62950      FALSE
aes-apu  7.062500     28.62950      FALSE
aes-asu 13.660714     29.63435      FALSE
an-anr   7.875000     28.62950      FALSE
an-ans  10.187500     28.62950      FALSE
an-apu  11.875000     28.62950      FALSE
an-asu  18.473214     29.63435      FALSE
anr-ans  2.312500     28.62950      FALSE
anr-apu 19.750000     28.62950      FALSE
anr-asu 26.348214     29.63435      FALSE
ans-apu 22.062500     28.62950      FALSE
ans-asu 28.660714     29.63435      FALSE
apu-asu  6.598214     29.63435      FALSE
```

Figura 6.18: Resultados do teste Kruskal-Wallis para o custo com base no nº de casos de teste adequados

6.4.2 Análise da Eficácia

Para a análise da segunda hipótese, relacionada à eficácia, foi utilizado o teste não paramétrico de Kruskal-Wallis em razão da suposição de normalidade dos erros ter sido violada, quando aplicado o teste de normalidade de Shapiro-Wilk. Dessa forma, os resultados do teste estão apresentados na Figura 6.19. A análise dessa tabela segue as mesmas instruções citadas na análise anterior. Para este resultado observa-se que na quarta coluna da

tabela que apresenta as diferenças para cada par de critério todos os resultados são *false*, significando que não é possível refutar a hipótese nula H_0, pois não foram observadas diferenças significantes entre ao menos um par de critérios. Assim sobre diferença de eficácia para os critérios nada pode ser afirmado.

```
Multiple comparison test after Kruskal-Wallis
p.value: 0.05
Comparisons
           obs.dif critical.dif difference
acu-ae   1.0833333     25.24885      FALSE
acu-aes  4.7500000     25.24885      FALSE
acu-an   8.3333333     25.24885      FALSE
acu-anr 16.7500000     25.24885      FALSE
acu-ans 17.4166667     25.24885      FALSE
acu-apu  0.5833333     25.24885      FALSE
acu-asu  6.7500000     25.24885      FALSE
ae-aes   5.8333333     25.24885      FALSE
ae-an    9.4166667     25.24885      FALSE
ae-anr  17.8333333     25.24885      FALSE
ae-ans  18.5000000     25.24885      FALSE
ae-apu   1.6666667     25.24885      FALSE
ae-asu   5.6666667     25.24885      FALSE
aes-an   3.5833333     25.24885      FALSE
aes-anr 12.0000000     25.24885      FALSE
aes-ans 12.6666667     25.24885      FALSE
aes-apu  4.1666667     25.24885      FALSE
aes-asu 11.5000000     25.24885      FALSE
an-anr   8.4166667     25.24885      FALSE
an-ans   9.0833333     25.24885      FALSE
an-apu   7.7500000     25.24885      FALSE
an-asu  15.0833333     25.24885      FALSE
anr-ans  0.6666667     25.24885      FALSE
anr-apu 16.1666667     25.24885      FALSE
anr-asu 23.5000000     25.24885      FALSE
ans-apu 16.8333333     25.24885      FALSE
ans-asu 24.1666667     25.24885      FALSE
apu-asu  7.3333333     25.24885      FALSE
```

Figura 6.19: Resultados do teste de Kruskal-Wallis para a eficácia dos critérios

6.4.3 Análise do Strength

Para análise da terceira hipótese, relacionada ao *strength*, foi usado o método estatístico de análise multivariada, em específico análise de cluster, pois, neste caso o objetivo era identificar as relações existentes entre os grupos analisados para detectar se há ou não relação de inclusão. Os dendrogramas foram construídos com base nas médias de cada grupo de critérios. Os números nas figuras representam os critérios, em que o número 1 representa o critério todos-nos, o número 2 o critério todos-nos-r, o 3 representa o critério todos-nos-s, o 4 representa o critério todas-arestas, o 5 representa o critério todas-arestas-s, o número 6 representa o critério todos-c-usos, o 7 representa o critério todos-p-usos e o número 8 representa o critério todos-s-usos. Cada nível da árvore representa um grupo com média similar.

Para o dendrograma que representa o conjunto adequado ao critério todos-nos (Figura 6.20) quando aplicado aos demais critérios, o nível inferior apresenta indícios de inclusão para o critérios todos-nos com relação aos critérios todos-nos-r e todos-nos-s. Com base nisso, a hipótese nula H_0 pode ser rejeitada.

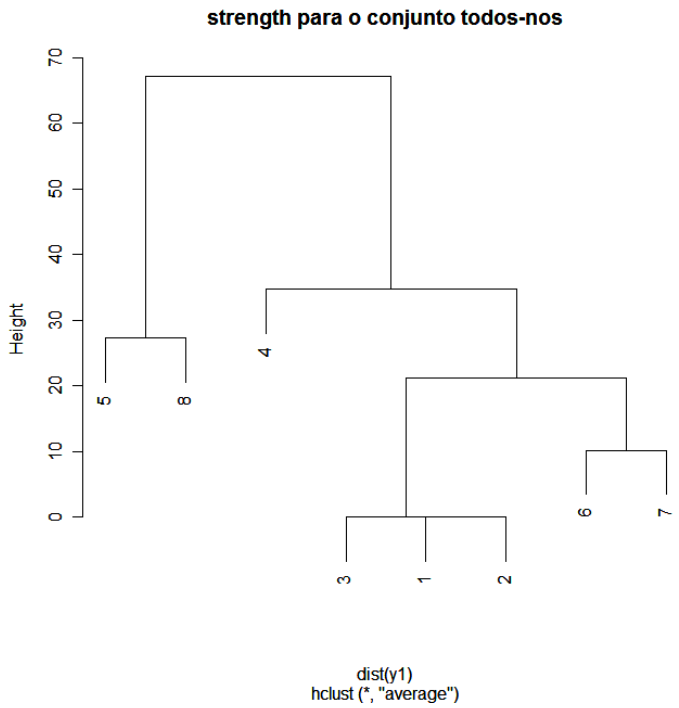

Figura 6.20: Dendrograma para o *strength* do conjunto todos-nos

Para o conjunto adequado ao critério todos-nos-r a análise de cluster identificou inclusão do critério todos-nos-s e vice-versa para o conjunto todos-nos-s que apresentou dendrograma igual, por isso este não foi apresentado aqui.

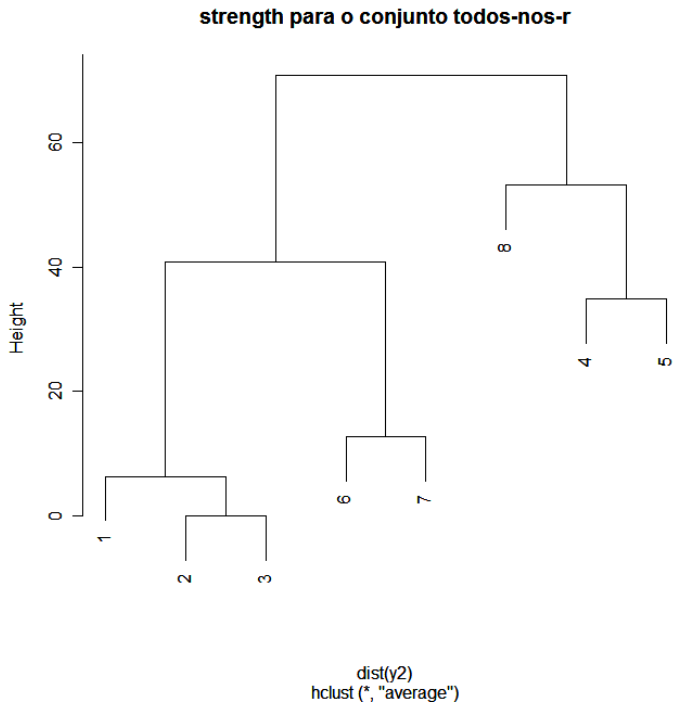

Figura 6.21: Dendrograma para o *strength* do conjunto todos-nos-r

Para o conjunto adequado ao critério todas-arestas (Figura 6.22) foi identificado que este pode incluir os critérios todos-nos, todos-nos-r, todos-nos-s e todas-arestas-s. Para o conjunto todas-arestas-s há indícios de inclusão dos critérios todos-nos-r e todos-nos-s.

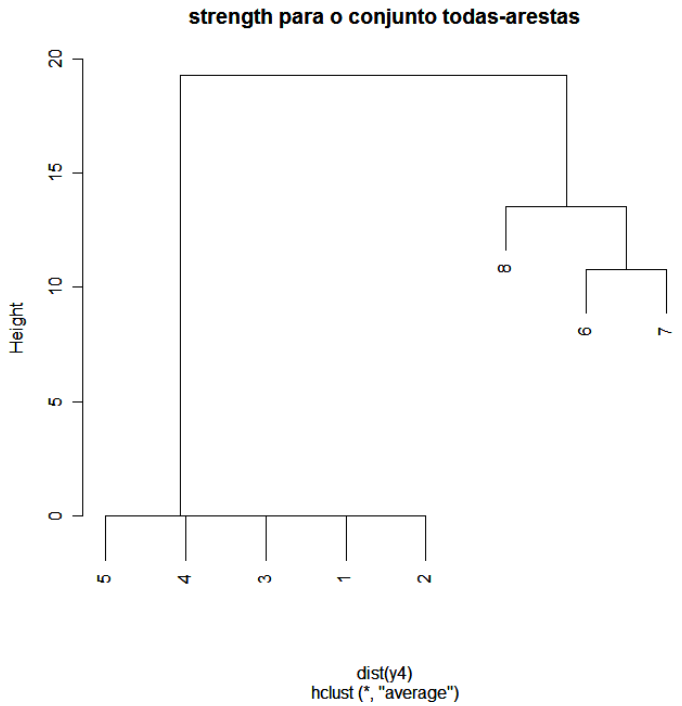

Figura 6.22: Dendrograma para o *strength* do conjunto todas-arestas

A Figura 6.23 apresenta a relação de inclusão para o conjunto todos-c-usos. Neste caso, o método identificou indícios de inclusão dos critérios todos-nos-r e todos-nos-s.

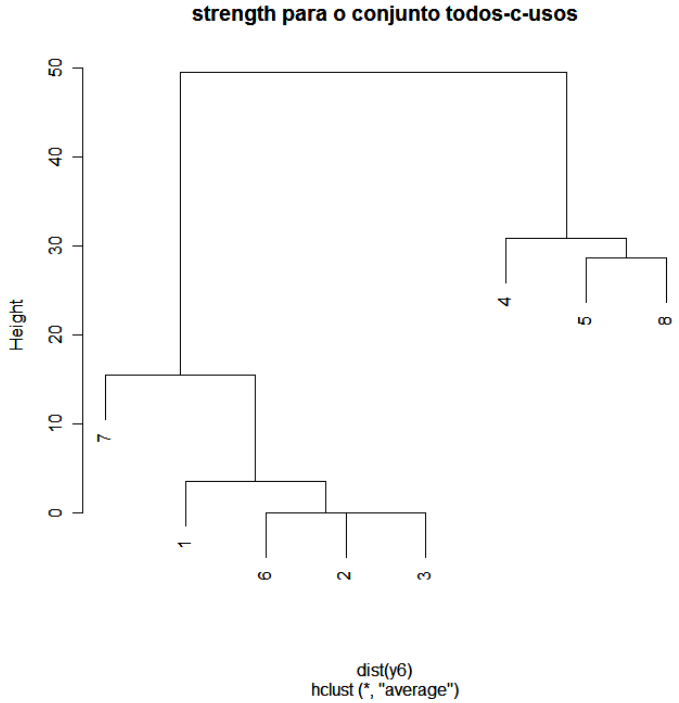

Figura 6.23: Dendrograma para o *strength* do conjunto todos-c-usos

Para o conjunto adequado ao critério todos-p-usos, foram identificados indícios de inclusão para os critérios todos-nos, todos-nos-r e todos-nos-s. E, finalmente, para o conjunto adequado ao critério todos-s-usos foram observados indícios de inclusão para os critérios todos-nos, todos-nos-r, todos-nos-s, todas-arestas e todas-arestas-s.

6.5 Síntese das Análises

Com base nos resultados apresentados e na análise de dados realizada estão como contribuições do trabalho para o teste de programas concorrentes:

1. Mais informações sobre o custo dos critérios, atestando que os critérios baseados no fluxo de controle e no fluxo de comunicação apresentam custos mais baixos que os critérios baseados em fluxo de dados e em passagem de mensagem. Assim os custos dos critérios com base no número de elementos requeridos em ordem crescente são: todos-c-usos, todos-nos, todos-p-usos, todas-arestas, todos-s-usos, todas-arestas-s, todos-nos-r e todos-nos-s. Com base no número de elementos não executáveis em ordem crescente são: todos-s-usos, todas-arestas-s, todas-arestas, todos-p-usos, todos-c-usos, todos-nos, todos-nos-r e todos-nos-s. Com base no número de casos de

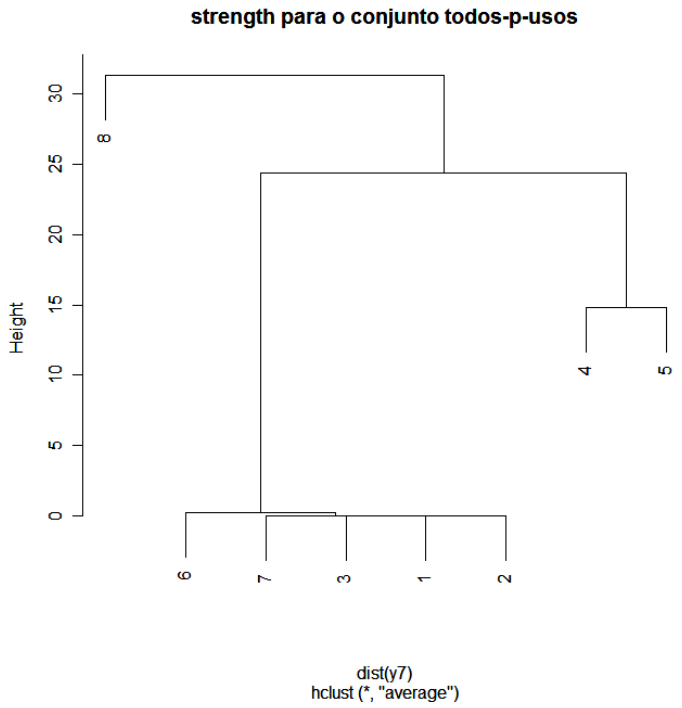

Figura 6.24: Dendrograma para o *strength* do conjunto todos-p-usos

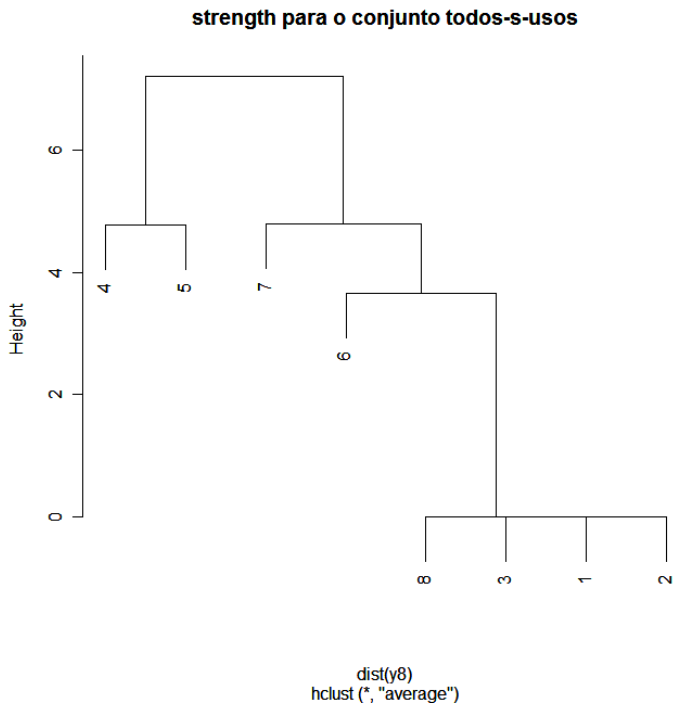

Figura 6.25: Dendrograma para o *strength* do conjunto todos-s-usos

teste adequados em ordem crescente são: todos-s-usos, todos-p-usos, todas-arestas, todos-c-usos, todas-arestas-s, todos-nos, todos-nos-r e todos-nos-s.

2. Mais informações sobre a eficácia de aplicação dos critérios. Considerando a ordem decrescente em eficácia estão: todos-s-usos, todos-c-usos, todos-p-usos, todas-arestas, todas-arestas-s, todos-nos, todos-nos-r e todos-nos-s.

3. Mais informações sobre o *strength* dos critérios. Há indícios de inclusão do critério todos-nos-r pelo critério todos-nos-s e vice-versa, indícios de inclusão dos critérios todos-nos-r e todos-nos-s pelos critérios todos-nos, todas-arestas, todas-arestas-s, todos-c-usos, todos-p-usos e todos-s-usos. Indícios de inclusão do critério todos-nos pelos critérios todas-arestas, todos-p-usos e todos-s-usos. Há indícios de inclusão do critério todas-arestas-s pelos critérios todas-arestas e todos-s-usos, bem como indícios sobre a inclusão do critério todas-arestas pelo critério todos-s-usos.

4. Os artefatos do experimento que podem ser acessados em <http://www.icmc.usp.br/~masbrit/artefatosExperimentoDissertacao.rar>.

Considerando esses resultados, uma estratégia de aplicação dos critérios poderia iniciar com a aplicação de critérios de menor custo e melhor eficácia e em seguida, aplicar critérios mais eficazes e de maior custo. Nesse sentido, uma possível estratégia de aplicação seria gerar casos de teste até atingir a cobertura máxima para o critério todos-c-usos (o qual inclui os critérios todos-nos-r e todos-nos-s). A partir do conjunto obtido na fase anterior, gerar casos de teste até atingir a cobertura máxima para os critérios todas-arestas (o qual inclui os critérios todas-arestas-s e todos-nos). Finalmente, adicionar novos casos de teste até atingir a cobertura máxima para os critérios todos-p-usos e todos-s-usos.

6.6 Considerações Finais

Neste capítulo foram apresentadas as análises dos dados obtidos com o experimento conduzido. Três hipóteses foram analisadas no decorrer do capítulo, resultando na proposição de uma estratégia mais geral de aplicação dos critérios de teste, capaz de incluir todos os critérios. Os resultados aqui apresentados fornecem subsídios para as conclusões do trabalho, apresentadas no próximo capítulo.

Capítulo

7

Conclusão

7.1 Caracterização da Pesquisa Realizada

Este trabalho teve por objetivo avaliar o custo, eficácia e *strength* dos critérios de teste estruturais definidos para programas concorrentes que usam o padrão de passagem de mensagem MPI. Para isso foi realizado um estudo experimental utilizando programas de vários domínios do contexto do estudo. Este trabalho complementa o estudo realizado por Hausen (2005) em que foi definida a ferramenta de teste ValiMPI. Observou-se que o custo dos critérios de teste baseados em fluxo de controle e no fluxo de comunicação são em geral menores que os custos dos critérios baseados em fluxo de dados e em passagem de mensagem. Com a análise usando estatística descritiva foi possível observar algumas relações entre a eficácia dos critérios, como exemplo, o critério todos-s-usos que parece ser o mais eficaz e os critérios todos-nos-r e todos-nos-s os menos eficazes. Porém considerando a eficácia, no teste de hipóteses nada pode ser afirmado. Sobre o *strength*, o critério todos-s-usos é o que possui mais alta dificuldade de satisfação enquanto os critérios todos-nos-r e todos-nos-s são os de menor dificuldade de satisfação. Nesse sentido foi estabelecida uma estratégia de aplicação dos critérios de teste que pode ser avaliada em trabalhos futuros.

7.2 Contribuições

Podem-se destacar como principais contribuições deste trabalho:

1. Resultados sobre a comparação de custo entre os critérios de teste.
2. Resultados sobre a comparação de eficácia entre os critérios de teste.
3. Resultados sobre a comparação de *strength* entre os critérios de teste.
4. A identificação de uma estratégia de teste com base nos resultados sobre custo, eficácia e *strength*.
5. A geração de um conjunto de artefatos contendo a documentação de toda atividade de teste que pode servir como material de auxílio ao ensino e treinamento de técnicas de testes de software.

7.3 Dificuldades e Limitações

Durante a execução do trabalho algumas dificuldades ocorreram. A primeira dificuldade encontrada foi durante a escolha dos programas que seriam utilizados no experimento, pois, não é tão comum encontrar *benchmarks* no contexto de programas concorrentes em *C* que usam o padrão de passagem de mensagem MPI. Além disso, as aplicações deveriam ter codificação que fossem adequadas às limitações da ferramenta de teste ValiMPI. Outra dificuldade encontrada foi a escassez de modelos de classificações de defeitos no contexto do trabalho. Apesar de existirem diversos estudos avaliando erros concorrentes, esses estudos são mais voltados ao paradigma de memória compartilhada. Mesmo com essas dificuldades, o estudo foi conduzido e espera-se que os artefatos gerados possam servir para a avaliação e comparação de novos critérios que venham a ser propostos para o apoio ao teste de programas concorrentes.

7.4 Trabalhos Futuros

Como trabalhos futuros decorrentes dessa dissertação destacam-se: a replicação desse estudo, considerando outros domínios de programas, defeitos e mais participantes envolvidos, a realização de um estudo teórico sobre os critérios de teste, a avaliação da estratégia de teste proposta, dentre outros.

Referências

Agrawal, H.; DeMillo, R. A.; Hathaway, B.; Hsu, W.; Hsu, W.; Krauser, E. W.; Martin, R. J.; Mathur, A. P.; Spafford, E. H. *Design of mutant operators for the C programming language.* Relatório Técnico, Software Engineering Research Center, Department of Computer Sciences, Purdue University, 1989.

Almasi, G. S.; Gottlieb, A. *Highly parallel computing.* Second ed. The Benjamin/-Cummings Publishing Company, Inc., 1994.

Andrews, G. R.; Schneider, F. B. Concepts and notations for concurrent programming. *ACM Computing Surveys*, v. 15, n. 1, p. 3–43, 1983.

Basili, V. R. The experimental paradigm in software engineering. In: *International Workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions*, 1993, p. 3–12.

Basili, V. R.; Selby, R. W.; Hutchens, D. H. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, v. 12, p. 733–743, 1986.

Bechini, A.; Cutajar, J.; Prete, C. A tool for testing of parallel and distributed programs in message-passing environments. In: *9th Mediterranean Electrotechnical Conference*, 1998, p. 1308 –1312 vol.2.

Beguelin, A. L. Xab: a tool for monitoring pvm programs. In: *26th Hawaii International Conference on System Sciences*, 1993, p. 102–103.

Birman, K. P.; Renesse, R. V. *Reliable distributed computing with the isis toolkit.* Los Alamitos, CA, USA: IEEE Computer Society Press, 1994.

Bonetti, D. F. *FPGAs em Problemas de Bioinformática.* Dissertação de Mestrado, Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação, ICMC-USP, 2010.

Brito, M. A. S.; Felizardo, K. R.; Souza, P. S. L.; Souza, S. R. S. *Concurrent software testing: A systematic review.* Relatório Técnico 359, Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação, ICMC-USP, 2010.

Carver, R.; Lei, Y. A general model for reachability testing of concurrent programs. In: *Formal Methods and Software Engineering*, 2004, p. 76–98.

Carver, R.; Tai, K. C. Deterministic execution testing of concurrent ada programs. In: *Conference on Tri-Ada 89: Ada technology in context: application, development, and deployment*, 1989, p. 528–544.

Chaim, M. L. *Poke-tool – uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados.* Dissertação de Mestrado, Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e Computação, FEEC/UNICAMP, Campinas, SP, 1991.

Chung, C.-M.; Shih, T. K.; Wang, Y.-H.; Lin, W.-C.; Kou, Y.-F. Task decomposition testing and metrics for concurrent programs. In: *7th International Symposium on Software Reliability Engineering*, 1996, p. 112–130.

Chusho, T. Test data selection and quality estimation based on the concept of essential branches for path testing. *IEEE Transactions on Software Engineering*, v. 13, n. 5, p. 509–517, 1987.

Coward, P. D. A review of software testing. *Information and Software Technology*, v. 30, n. 3, p. 189–198, 1988.

Damodaran-Kamal, S. K.; Francioni, J. M. Nondeterminancy: testing and debugging in message passing parallel programs. In: *ACM/ONR workshop on Parallel and distributed debugging*, 1993, p. 118–128.

Delamaro, M. E.; Maldonado, J. C.; Jino, M. *Introdução ao teste de software*, v. 394 de *Campus*. Elsevier, 2007.

DeMillo, R. A. *Software testing and evaluation.* Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1986.

DeSouza, J.; Kuhn, B.; de Supinski, B. R.; Samofalov, V.; Zheltov, S.; Bratanov, S. Automated, scalable debugging of mpi programs with intel message checker. In: *2th*

International workshop on Software Engineering for high performance computing system applications, 2005, p. 78–82.

Dória, E. *Replicação de estudos empíricos em engenharia de software.* Dissertação de Mestrado, Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação, ICMC-USP, São Carlos, SP, 2001.

Edelstein, O.; Farchi, E.; Goldin, E.; Nir, Y.; Ratsaby, G.; Ur, S. Framework for testing multi-threaded java programs. *Concurrency Computation Practice and Experience*, v. 15, n. 3-5 SPEC., p. 485–499, 2003.

Endo, A. T.; Simão, A. S.; Souza, S. R. S.; Souza, P. S. L. Web services composition testing: A strategy based on structural testing of parallel programs. In: *Testing: Academic and Industrial Conference - Practice and Research Techniques*, 2008, p. 3–12.

Foster, I. *Designing and building parallel programs: Concepts and tools for parallel software engineering.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

Frankl, P. G.; Weiss, S. N. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, v. 19, p. 774–787, 1993.

Ghezzi, C.; Jazayeri, M. *Programming languages concepts.* 2 ed. New York: John Wiley and Sons, 1987.

Gligoric, M.; Jagannath, V.; Marinov, D. MuTMuT: Efficient exploration for mutation testing of multithreaded code. In: *Third International Conference on Software Testing, Verification and Validation*, 2010, p. 55 –64.

Grama, A.; Karpys, G.; Kumar, V.; Gupta, A. *Introduction to parallel computing.* Second ed. Addison Wesley, 2003.

Harvey, C.; Strooper, P. Testing java monitors through deterministic execution. In: *13th Australian Conference on Software Engineering*, 2001, p. 61–67.

Hausen, A. C. *ValiMPI : uma ferramenta de teste estrutural para programas paralelos em ambiente de passagem de mensagem.* Mestrado in Informática, Setor de Ciencias Exatas, Programa de Pós-Graduação em Informática – Universidade Federal do Paraná, UFPR, 2005.

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

Hetzel, W. C. *An experimental analysis of program verification methods.* Tese de Doutoramento, Dept. of Computer Science, Univ. of North Carolina, Chapel Hill, 1976.

Howden, W. E. Methodology for the generation of program test data. *IEEE Transactions on Computers*, v. 24, n. 5, p. 554–560, 1975.

Howden, W. E. Theoretical and empirical studies of program testing. In: *3rd international conference on Software engineering*, 1978, p. 305–311.

Hwang, K.; Briggs, F. A. *Computer architecture and parallel processing.* McGraw-Hill College, 1984.

IEEE. *IEEE standard glossary of Software Engineering terminology.* Padrão 620.12, IEEE, 1990.

Jagannath, V.; Gligoric, M.; Lauterburg, S.; Marinov, D.; Agha, G. Mutation operators for actor systems. In: *Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010, p. 157 –162.

Joshi, P.; Naik, M.; Park, C.-S.; Sen, K. Calfuzzer: An extensible active testing framework for concurrent programs. In: *21st International Conference on Computer Aided Verification*, 2009, p. 675–681.

Kamsties, E.; Lott, C. M. An empirical evaluation of three defect-detection techniques. In: *5th European Software Engineering Conference*, 1995, p. 362–383.

Koppol, P.; Carver, R.; Tai, K.-C. Incremental integration testing of concurrent programs. *IEEE Transactions on Software Engineering*, v. 28, n. 6, p. 607–623, 2002.

Koppol, P. V.; Tai, K.-C. An incremental approach to structural testing of concurrent software. In: *ACM SIGSOFT international symposium on Software testing and analysis*, 1996, p. 14–23.

Krawczyk, H.; Wiszniewski, B. *Classification of software defects in parallel programs.* Relatório Técnico, Faculty of Eletronics, Technical University of Gdansk, Poland, 1994.

Krawczyk, H.; Wiszniewski, B. *A method for determining testing scenarios for parallel and distributed software.* Relatório Técnico, n 193, 1996.

Laski, J. W.; Korel, B. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, v. 9, n. 3, p. 347–354, 1983.

Lei, Y.; Carver, R. Reachability testing of semaphore-based programs. In: *28th Annual International Computer Software and Applications Conference*, 2004, p. 312–317.

Lei, Y.; Carver, R. H. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, v. 32, n. 6, p. 382–403, 2006.

Lei, Y.; Carver, R. H.; Kacker, R.; Kung, D. A combinatorial testing strategy for concurrent programs. *Software Testing Verification and Reliability*, v. 17, n. 4, p. 207–225, 2007.

Li, S. Q.; Chen, H. Y.; Sun, Y. X. A framework of reachability testing for Java multithread programs. In: *IEEE International Conference on Systems, Man and Cybernetics*, 2004, p. 2730–2734.

Liang, Y.; Li, S.; Zhang, H.; Han, C. Timing-sequence testing of parallel programs. *Journal of Computer Science and Technology*, v. 15, n. 1, p. 84 – 95, 2000.

Linkman, S.; Vincenzi, A. M. R.; Maldonado, J. An evaluation of systematic functional testing using mutation testing. In: *7th International Conference on Empirical Assessment in Software Engineering*, 2003, p. 1–15.

Lourenço, J.; Cunha, J.; Krawczyk, H.; Kuzora, P.; Neyman, M.; Wiszniewski, B. An integrated testing and debugging environment for parallel and distributed programs. In: *23rd EUROMICRO Conference 'New Frontiers of Information Technology'*, 1997, p. 291–298.

Machado, M. C. C. *Estudo e definição de mecanismos para redução do custo de aplicação do teste de programas concorrentes.* Dissertação de Mestrado, Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação, ICMC-USP, 2011.

Maldonado, J. C. *Critérios potenciais usos: Uma contribuição ao teste estrutural de software.* Tese de doutoramento, Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e Computação, FEEC/UNICAMP, Campinas, SP, 1991.

Maldonado, J. C.; Barbosa, E. F.; ; Vincenzi, A. M. R.; Delamaro, M. E.; Souza, S. R. S.; Jino, M. *Introdução ao teste de software.* Instituto de Ciências Matemáticas e de Computação - ICMC-USP, nota Didática n. 65, 2004.

Maldonado, J. C.; Vergilio, S. R.; Chaim, M. L.; Jino, M. Critérios potenciais usos: Análise da aplicação de um benchmark. In: *VI Simpósio Brasileiro de Engenharia de Software*, 1992, p. 357–374.

Malevris, N.; Yates, D. F. The collateral coverage of data flow criteria when branch testing. *Information and Software Technology*, v. 48, n. 8, p. 676 – 686, 2006.

Mathur, A. P.; Wong, W. E. Evaluation of the cost of alternative mutation strategies. In: *7th Simpósio Brasileiro de Engenharia de Software*, 1993, p. 320–335.

Mathur, A. P.; Wong, W. E. An empirical comparison of data flow and mutation based test adequacy criteria. *Software Testing, Verification and Reliability*, v. 4, n. 1, p. 9–31, 1994.

Miller, R. E. Comparison of some theoretical models of parallel computation. *IEEE Transactions on Computers*, v. C-22, n. 8, p. 710–717, 1973.

Myers, G. J. A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, v. 21, n. 9, p. 760–768, 1978.

Myers, G. J.; Sandler, C.; Badgett, T.; Thomas, T. M. *The art of software testing.* John Wiley & Sons, Inc., Hoboken, New Jersey, 2004.

Offutt, A. J.; Lee, A.; Rothermel, G.; Untch, R. H.; Zapf, C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, v. 5, p. 99–118, 1996a.

Offutt, A. J.; Pan, J.; Tewary, K.; Zhang, T. An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, v. 26, n. 2, p. 165–176, 1996b.

Ostrand, T. J.; Balcer, M. J. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, v. 31, n. 6, p. 676–686, 1988.

Park, M.-Y.; Shim, S. J.; Jun, Y.-K.; Park, H.-R. MPIRace-check: detection of message races in MPI programs. In: *2nd international conference on Advances in grid and pervasive computing*, 2007, p. 322–333.

Pressman, R. S. *Software engineering: Practitioner's approach.* 6 ed McGraw-Hill, 2005.

Pu, F.; Xu, H.-Y. A feasible strategy for reachability testing of internet-based concurrent programs. In: *IEEE International Conference on Networking, Sensing and Control*, 2008, p. 1559 –1564.

Quinn, M. J. *Parallel programming in C with MPI and OpenMP.* McGraw-Hill Education Group, 2003.

Rapps, S.; Weyuker, E. J. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, v. 11, n. 4, p. 367–375, 1985.

Saini, G. *An empirical evaluation of adequacy criteria for testing concurrent programs.* Dissertação de Mestrado, Faculty of the Graduate School of The University of Texas at Arlington, 2005.

Sant'Ana, T. D. *Ambiente de simulação e teste de programas paralelos.* Dissertação de Mestrado, Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação, ICMC-USP, 2001.

Sarmanho, F. S.; Souza, P. S. L.; Souza, S. R. S.; Simão, A. S. Structural testing for semaphore-based multithread programs. In: *8th international conference on Computational Science*, 2008, p. 337–346.

Sen, A.; Abadir, M. Coverage metrics for verification of concurrent SystemC designs using mutation testing. In: *IEEE International High Level Design Validation and Test Workshop*, 2010, p. 75–81.

Seo, H.-S.; Chung, I. S.; Kim, B. M.; Kwon, Y. R. The design and implementation of automata-based testing environment for Java multi-thread programs. In: *Eighth Asia-Pacific on Software Engineering Conference*, 2001, p. 221–228.

Simão, A. S.; Vincenzi, A. M. R.; Santana, A. C. L. A language for the description of program instrumentation and automatic generation of instrumenters. *CLEI Electronic Journal*, v. 6, n. 1, 2003.

Sjoberg, D. I. K.; Dyba, T.; Jorgensen, M. The future of empirical methods in software engineering research. In: *Future of Software Engineering*, 2007, p. 358–378.

Snir, M.; Otto, S.; Steven, H.; Walker, D.; Dongarra, J. *Mpi: The complete reference.* Relatório Técnico, MIT Press, Massachussets, 1996.

Souza, P. L.; Sawabe, E. T.; Simão, A. S.; Vergilio, S. R.; Souza, S. R. S. ValiPVM - a graphical tool for structural testing of PVM programs. In: *15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2008a, p. 257–264.

Souza, S.; Maldonado, J. C. Avaliação do impacto da minimização de conjuntos de casos de teste no custo e eficácia do critério análise de mutantes. In: *XI Simpósio Brasileiro de Engenharia de Software*, 1997.

Souza, S.; Maldonado, J. C.; Vergilio, S. R. Análise de mutantes e potenciais-usos: Uma avaliação empírica. In: *Conferencia internacional de Tecnologia de Software*, 1997, p. 225–236.

Souza, S. R. S. *Avaliação do custo e eficácia do critério análise de mutantes na atividade de teste de software.* Dissertação de Mestrado, Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação, ICMC-USP, São Carlos, SP, 1996.

Souza, S. R. S.; Brito, M. A. S.; Silva, R. A.; Souza, P. S. L.; Zaluska, E. Research in concurrent software testing: A systematic review. In: *IX Workshop on Parallel and Distributed Systems: Testing, Analysis and Debugging - International Symposium on Software Testing and Analysis*, 2011, p. 1–5.

Souza, S. R. S.; Fabbri, S. C. P. F.; Barbosa, E. F.; Chaim, M. L.; Vincenzi, A. M. R.; Delamaro, M. E.; Jino, M.; Maldonado, J. C. *Introdução ao teste de software*, v. 1, cáp. Estudos Teóricos e Experimentais M. E. Delamaro and J. C. Maldonado and M. Jino, p. 251–268, 2007a.

Souza, S. R. S.; Vergilio, R.; Souza, P. S. L.; Simão, S.; Hausen, A. C. Structural testing criteria for message-passing parallel programs. *Concurrency Computation Practice and Experience*, v. 20, n. 16, p. 1893–1916, 2008b.

Souza, S. R. S.; Vergilio, S. R.; Souza, P. S. L. *Introdução ao teste de software*, v. 1, cáp. Teste de Programas Concorrentes M. E. Delamaro and J. C. Maldonado and M. Jino, p. 231–249, 2007b.

Souza, S. R. S.; Vergilio, S. R.; Souza, P. S. L.; Simão, A. S.; Goncalves, T. B.; Lima, A. M.; Hausen, A. C. ValiPar: A testing tool for message-passing parallel programs. In: *International Conference on Software Engineering and Knowledge Engineering*, 2005, p. 386–391.

Stoller, S. Testing concurrent java programs using randomized scheduling. In: *Second Workshop on Runtime Verification*, 2002, p. 149–164.

Tanenbaum, A. S. *Modern operating systems.* 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2001.

Taylor, R.; Levine, D.; Kelly, C. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, v. 18, n. 3, p. 206–215, 1992.

Taylor, R. N. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, v. 26, n. 5, p. 361–376, 1983.

Travassos, G. H. *Introdução à engenharia de software experimental.* Relatório Técnico, Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia, Universidade Federal do Rio de Janeiro, COPPE/UFRJ, RJ-Brasil, 2002.

Ural, H.; Yang, B. A structural test selection criterion. *Information Processing Letters*, v. 28, n. 3, p. 157–163, 1988.

Vergilio, S. R.; Maldonado, J. C.; Jino, M. Caminhos não executáveis na automatização das atividades de teste. In: *Simpósio Brasileiro de Engenharia Software*, 1992, p. 343–356.

Vergilio, S. R.; Souza, S. R. S.; Souza, P. S. L. Coverage testing criteria for message passing parallel programs. In: *6th IEEE Latin-American Test Workshop*, 2005, p. 161–166.

Vilela, P. R.; Maldonado, J. C.; Jino, M. Program graph visualization. *Software-Practice & Experience*, v. 27, n. 11, p. 1245–1262, 1997.

Weyuker, E. J. The cost of data flow testing: An empirical study. *IEEE Transactions on Software Engineering*, v. 16, n. 2, p. 121–128, 1990.

Wildman, L.; Long, B.; Strooper, P. Dealing with non-determinism in testing concurrent Java components. In: *12th Asia-Pacific Software Engineering Conference*, 2005, p. 393–400.

Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M. C.; Regnell, B.; Wesslén, A. *Experimentation in software engineering: an introduction.* Norwell, MA, USA: Kluwer Academic Publishers, 2000.

Wong, W.E., L. Y. Reachability graph-based test sequence generation for concurrent programs. *International Journal of Software Engineering and Knowledge Engineering*, v. 18, n. 6, p. 803–822, 2008.

Wong, W. E. *On mutation and data flow.* Tese de Doutoramento, West Lafayette, IN, USA, 1993.

Wong, W. E.; Delamaro, M. E.; Maldonado, J. C.; Mathur, A. P. Constrained mutation in c programs. In: *8th Brazilian Symposium on Software Engneering*, 1994, p. 439–452.

Wong, W. E.; Mathur, A. P.; Maldonado, J. C. Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness. In: *Software Quality and Productivity: Theory, practice and training*, 1995, p. 258–265.

Wood, M.; Roper, M.; Brooks, A.; Miller, J. Comparing and combining software defect detection techniques: a replicated empirical study. In: *6th European Software Engineering conference*, 1997, p. 262–277.

Xufang Gong, Yanchen Wang, Y. Z. B. L. On testing multi-threaded java programs. In: *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 2007, p. 702–706.

Yang, C.-S.; Pollock, L. All-uses testing of shared memory parallel programs. *Software Testing Verification and Reliability*, v. 13, n. 1, p. 3–24, 2003.

Yang, C.-S.; Pollock, L. L. The challenges in automated testing of multithreaded programs. In: *14th International Conference on Testing Computer Software*, 1997, p. 157–166.

Yang, C.-S. D.; Souter, A. L.; Pollock, L. L. All-du-path coverage for parallel programs. In: *ACM SIGSOFT international symposium on Software testing and analysis*, 1998, p. 153–162.

Yang, R.-D.; Chung, C.-G. Path analysis testing of concurrent programs. *Information and Software Technology*, v. 34, n. 1, p. 43–56, 1992.

Young, M.; Taylor, R. N.; Levine, D. L.; Forester, K.; Brodbeck, D. *A concurrency analysis tool suite: Rationale, design, and preliminary experience.* Relatório Técnico, 1992.