Geração automática de dados de teste para
programas concorrentes com meta-heurística

José Dario Pintor da Silva

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 21/11/2014

Assinatura: _____

Geração automática de dados de teste para programas concorrentes com meta-heurística

José Dario Pintor da Silva

***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
Novembro de 2014

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

D586g Dario Pintor da Silva, José
 Geração automática de dados de teste para
programas concorrentes com meta-heurística / José
Dario Pintor da Silva; orientadora de Souza Simone
do Rocio Senger. -- São Carlos, 2014.
 80 p.

 Tese (Doutorado - Programa de Pós-Graduação em
Matemática) -- Instituto de Ciências Matemáticas e
de Computação, Universidade de São Paulo, 2014.

 1. Geração automática de Dados . 2. Teste de
Software. 3. Programação concorrente. 4. Meta-
heurística. I. Simone do Rocio Senger, de Souza,
orient. II. Título.

A programação concorrente é cada vez mais utilizada nos sistemas atuais com o objetivo de reduzir custos e obter maior eficiência no processamento. Com a importância da programação concorrente é imprescindível que programas que implementam esse paradigma apresentem boa qualidade e estejam livres de defeitos. Assim, diferentes técnicas e critérios de teste vêm sendo definidos para apoiar a validação de aplicações desenvolvidas nesse paradigma. Nesse contexto, a geração automática de dados de teste é importante, pois permite reduzir o custo na geração e seleção de dados relevantes. O uso de técnicas meta-heurísticas tem sido uma área de grande interesse entre os pesquisadores para geração de dados, pois essas técnicas apresentam abordagens aplicáveis a problemas complexos e de difícil solução. Considerando esse aspecto, este trabalho apresenta uma abordagem de geração automática de dados para o teste estrutural de programas concorrentes em MPI (*Message Passing Interface*). A meta-heurística usada foi Algoritmo Genético em que a busca é guiada por critérios de teste que consideram características implícitas de programas concorrentes. O desempenho da abordagem foi avaliado por meio da cobertura dos dados de testes, da eficácia em revelar defeitos e do custo de execução. Para comparação, a geração aleatória foi considerada. Os resultados indicaram que é promissor usar geração de dados de teste no contexto de programas concorrentes, com resultados interessantes em relação à eficácia e cobertura dos requisitos de teste.

Abstract

Concurrent programming has been increasingly used in current systems in order to reduce costs and obtain higher processing efficiency and, consequently, it is expected that these systems have high quality. Therefore, different techniques and testing criteria have been proposed aiming to support the verification and validation of the concurrent applications. In this context, the automated data test generation allows to reduce the testing costs during the generation and selection of data tests. Metaheuristic technique has been widely investigated to support the data test generation because this technique has presented good results to complex and costly problems. In this work, we present an approach to the automated data test generation for message-passing concurrent programs in MPI (Message Passing Interface). The generation of data test is performed using the genetic algorithm metaheuristic technique, guiding by structural testing criteria. An experimental study was conducted to evaluate the proposed approach, analyzing the effectiveness and application cost. The results indicate that the genetic algorithm is a promising approach to automated test data generation for concurrent programs, presenting good results in relation to effectiveness and data test coverage.

Sumário

1	Introdução	1
1.1	Contextualização	1
1.2	Motivação	3
1.3	Objetivos	3
1.4	Organização	3
2	Fundamentação Teórica	4
2.1	Considerações Iniciais	4
2.2	Teste de Software	4
2.2.1	Técnicas e Critérios de Teste	7
2.3	Programação Concorrente	10
2.3.1	Paradigma de Memória Compartilhada	10
2.3.2	Passagem de Mensagem	12
2.3.3	Teste Estrutural de Programas Concorrentes	16
2.3.3.1	Ferramenta de Teste ValiMPI	21
2.4	Técnicas para Geração de Dados de Teste	23
2.4.1	Geração Aleatória	23
2.4.2	Geração com Execução Simbólica	24
2.4.3	Geração com Execução Dinâmica	24
2.4.3.1	Geração com Técnicas de Computação Evolutiva	24
2.4.3.2	Subida da Encosta (<i>Hill Climbing</i>)	25
2.4.3.3	Têmpera Simulada (<i>Simulated Annealing</i>)	26
2.4.3.4	Algoritmo Genéticos	27

2.5	Considerações Finais	32
3	Geração de Dados de Teste Usando Meta-heurística	33
3.1	Considerações Iniciais	33
3.2	Trabalhos Sobre Geração de Dados de Teste	33
3.3	Considerações Finais	38
4	Abordagem de Geração de Dados Proposta	39
4.1	Considerações Iniciais	39
4.2	Apresentação da Abordagem Proposta	40
4.3	Arquitetura e Funcionamento da ValiGen	44
4.3.1	Arquivo de Saída	49
4.3.2	Exemplo de Uso	50
4.4	Considerações Finais	51
5	Avaliação Experimental da Abordagem Proposta	52
5.1	Considerações Iniciais	52
5.2	Definição do Experimento	53
5.2.1	Objetivo	53
5.2.2	Metas	53
5.3	Planejamento	54
5.3.1	Seleção do Contexto	54
5.3.2	Hipóteses	54
5.3.3	Seleção dos Indivíduos	55
5.3.4	Variáveis	56
5.3.5	Descrição da Instrumentação do Experimento	56
5.3.6	Ameaça à Validade	57
5.4	Preparação e Execução	57
5.4.1	Ambiente de Testes	58
5.4.2	Descrição dos Programas	58
5.4.3	Execução	58
5.5	Análise dos Resultados	60
5.5.1	Análise do percentual de cobertura	60
5.5.2	Análise da eficácia	61

5.5.3	Análise do custo em relação ao tempo de execução	62
5.5.4	Análise das Hipóteses	63
5.6	Considerações Finais	64
6	Conclusão e Trabalhos Futuros	65
6.1	Caracterização da Pesquisa Realizada	65
6.2	Contribuições	66
6.3	Dificuldades e Limitações	66
6.4	Trabalhos Futuros	66
	Referências	74
A	Apêndice	75
A.1	Arquivo fonte do programa <i>GCD</i>	75
A.2	Arquivo fonte do programa <i>Mmult</i>	77

Lista de Figuras

2.1	Cenário Típico da Atividade de Teste (Delamaro et al., 2007b).	6
2.2	Exemplo do GFC (Grafo de Fluxo de Controle) para o programa Identifier (Barbosa et al., 2007)	8
2.3	Código fonte do programa GCD em MPI	15
2.4	Arquitetura da ferramenta de teste ValiMPI (Souza et al., 2008b).	21
2.5	Algoritmo de busca de Subida da Encosta (Russell e Norvig, 2009).	25
2.6	Algoritmo de Têmpera Simulada (Russell e Norvig, 2009).	26
2.7	Um cromossomo codificado em binário.	27
2.8	Método de seleção por Roleta.	29
2.9	Método de seleção por Torneio com n igual a três.	29
2.10	Representação de cruzamento com um ponto de corte.	30
2.11	Exemplificação de um operador de mutação.	30
2.12	Pseudocódigo de um Algoritmo Genético (Russell e Norvig, 2009).	31
4.1	Os indivíduos são representados por uma concatenação de blocos e cada bloco está relacionado a uma entrada do programa.	41
4.2	Não determinismo na geração de dados.	43
4.3	Arquitetura do módulo ValiGen.	45
4.4	Arquivo de configuração usado no exemplo.	46
4.5	Arquivo de cobertura dos indivíduos de uma população.	49
4.6	Arquivo de cobertura dos elementos requeridos.	50
5.1	Tipos de defeitos inseridos nos programas	55
5.2	Representação gráfica da cobertura do programa <i>GCD</i>	61

5.3	Representação gráfica da cobertura do programa <i>Mmult</i>	61
5.4	Representação gráfica da eficácia do programa <i>GCD</i>	61
5.5	Representação gráfica da eficácia do programa <i>Mmult</i>	62
5.6	Representação gráfica do custo de execução no programa <i>GCD</i>	62
5.7	Representação gráfica do custo de execução no programa <i>Mmult</i>	63

Introdução

1.1 Contextualização

O teste de software é de suma importância, dentro do processo de desenvolvimento, para a garantia de qualidade de software, e tem como objetivo revelar a presença de defeitos no produto em desenvolvimento (Pressman, 2010). A geração de dados de teste — que consiste em identificar dados de entrada que satisfaçam um determinado critério de teste e que sejam capazes de revelar defeitos ainda não revelados — tem como principal objetivo a redução de custo e esforço de desenvolvimento.

Embora a geração automática de dados de teste seja muito desejada, não existe um algoritmo de propósito geral para determinar um conjunto de teste que satisfaça um critério. Nem mesmo é possível determinar automaticamente se esse conjunto existe (Vergilio et al., 2007). Apesar de ser um problema indecidível (McMinn, 2004), existem muitas técnicas que podem ser usadas para geração de dados de teste para cobrir os diversos critérios existentes, sendo as mais conhecidas: geração aleatória; geração com execução simbólica; geração com execução dinâmica e geração que usa técnicas da computação evolutiva (Koirala e Sheikh, 2009; Patton, 2005; Vergilio et al., 2007).

Recentemente, uma classe de técnicas para geração de dados tem recebido bastante atenção por parte dos pesquisadores que consiste em aplicar otimização baseada em busca, conhecida como Teste Baseado em Busca (do inglês *Search-Based Testing*). O Teste Baseado em Busca é o processo de gerar automaticamente dados de teste de acordo com um critério de adequação (função de aptidão) usando algoritmos de otimização baseado em busca que é guiado por uma função de avaliação (Harman e McMinn, 2010). Essas técnicas aplicadas para o contexto de programas sequenciais (Bueno et al., 2001; Hrubá et al., 2012; Krena et al., 2011; Letko, 2010), apresentam resultados motivadores. Entretanto, poucos trabalhos exploram a geração de dados para o contexto de programas concorrentes.

A programação concorrente está tornando-se cada vez mais popular no desenvolvimento de softwares modernos. Pode-se citar como exemplo um servidor Web que cria processos (ou *threads*) separados para atender as requisições dos clientes, aumentando com isso, o desempenho necessário das aplicações. Esse paradigma de desenvolvimento é essencial para a construção de aplicações capazes de reduzir o tempo computacional em vários domínios, tais como: previsão de tempo, dinâmica de fluídos, processamento de imagens, química quântica entre outros. Esses programas apresentam novas características como comunicação, sincronização e não determinismo que devem ser consideradas durante a proposição de mecanismos de apoio para o desenvolvimento dessas aplicações.

Diferentemente dos programas tradicionais, a computação distribuída envolve programas paralelos que interagem para realizar as tarefas. Essa interação pode ocorrer de forma sincronizada ou não, em que esses programas (ou processos) podem ou não concorrerem pelos mesmos recursos computacionais. A construção de processos concorrentes requer o uso de algumas primitivas fundamentais, necessárias para permitir: 1) a definição de quais processos serão executados em paralelo; 2) a iniciação e finalização dos processos concorrentes e 3) a coordenação entre os processos concorrentes enquanto estes estiverem executando (Almasi e Gottlieb, 1994; McMinn, 2004; Wong et al., 2005). Desses três tipos de primitivas, destacam-se as primitivas necessárias à interação (comunicação e sincronização) entre os processos, devido à sua frequência de utilização bem como impacto no desempenho final do programa concorrente.

A atividade de teste no contexto de programas concorrentes é considerada mais complexa. Além das dificuldades inerentes à atividade de teste, outras ocorrem devido, principalmente, ao comportamento não determinístico dessas aplicações. Múltiplas execuções de um programa concorrente com a mesma entrada podem executar diferentes sequências de sincronização e podem produzir diferentes resultados corretos.

Cabe à atividade de teste, nesse cenário, identificar se todas as sequências de sincronização possíveis foram executadas e se as saídas obtidas estão corretas. Essa característica difere os programas concorrentes dos programas sequenciais e precisa ser considerada durante a atividade de teste de software. Nesse sentido, Yang e Pollock (2003) apresentam os principais desafios impostos ao teste de programas concorrentes: 1) necessidade de desenvolver técnicas de análise estática apropriadas para programas concorrentes; 2) detectar situações não desejadas como: erros de sincronização, comunicação, de execução com a mesma entrada de teste e forçar a execução de um caminho na presença de não determinismo; 4) gerar uma representação do programa concorrente que capture informações pertinentes ao teste; 5) investigar a aplicação de critérios de teste sequenciais para programas concorrentes; e 6) projetar critérios de fluxo de dados para programas concorrentes, considerando troca de mensagens e variáveis compartilhadas.

Os critérios de teste atuais, no contexto de programas concorrentes, são utilizados para guiar o testador na seleção dos dados de teste. Assim, a ideia deste trabalho é complementar essa funcionalidade, oferecendo uma abordagem que automatize a geração desses dados de teste.

1.2 Motivação

As seguintes questões serviram de motivação para a proposição deste trabalho:

1. A geração de dados é reconhecida como uma atividade importante dentro da fase de teste, pois permite reduzir o custo na seleção de dados relevantes;
2. Técnicas meta-heurísticas se mostram uma boa opção para a resolução do problema de indecidibilidade inerente à geração de dados;
3. Inexistência de trabalhos que abordam geração de dados no contexto de programas concorrentes que usam meta-heurísticas;

1.3 Objetivos

O uso de técnicas meta-heurísticas tem sido uma área de grande interesse entre os pesquisadores para geração de dados, pois são vistas como soluções para problemas indecidíveis a um custo e tempo aceitável. Nesse contexto, o objetivo deste projeto de mestrado é propor uma abordagem para geração automática de dados de teste para programas concorrentes com o uso da meta-heurística Algoritmo Genético, que deverá apoiar a aplicação de critérios de teste nesse domínio, capaz de auxiliar na identificação de defeitos comuns relacionados com essas aplicações, como *deadlock*, *starvation* e erros de sincronização.

1.4 Organização

No Capítulo 2, são apresentados os principais conceitos e terminologias, técnicas e critérios da área de teste de software e também são abordados os principais conceitos de programas concorrentes e testes para estes programas. Ainda nesse capítulo, é apresentada uma visão geral das principais técnicas existentes para geração automática de dados de teste, com um foco maior na geração usando a meta-heurística Algoritmo Genético, que será utilizada neste trabalho. No Capítulo 3, são apresentados os trabalhos relacionados a essa proposta de mestrado que foram a base da fundamentação teórica desse trabalho. No Capítulo 4, é apresentada a abordagem para geração de dados proposta e a descrição de um módulo que implementa a abordagem. No Capítulo 5, é apresentado um estudo experimental para validação da abordagem apresentada. Finalmente, no Capítulo 6, é apresentada a conclusão dessa dissertação de mestrado.

Fundamentação Teórica

2.1 Considerações Iniciais

Neste capítulo são apresentados os principais temas relacionados à fundamentação teórica desse projeto mestrado. Ao longo do capítulo são apresentados conceitos relacionados aos seguintes temas: Teste de Software, Programação Concorrente e Técnicas para Geração de Dados de Teste. Este conhecimento é necessário para melhor entendimento dos capítulos seguintes.

O capítulo está organizado da seguinte forma: na Seção 2.2 são apresentados os principais conceitos e terminologias da área de teste de software e também técnicas e critérios de teste. Na Seção 2.3 são apresentados os principais conceitos de programas concorrentes e seus paradigmas, o padrão MPI, os principais conceitos de teste para programas concorrentes e a ferramenta de teste ValiMPI. Na Seção 2.4 é apresentada uma visão geral das principais técnicas existentes para geração de dados de teste com um foco maior na geração com Algoritmo Genético, pois será utilizada no escopo deste trabalho.

2.2 Teste de Software

A atividade de teste é apontada como uma das mais onerosas no desenvolvimento de software. Por esse motivo, e pela percepção de pesquisadores e engenheiros de que qualidade é um fator essencial no desenvolvimento de software, muitos trabalhos têm sido publicados na área de teste de software (Ferguson e Korel, 1996; Pinto e Vergilio, 2010; Sarmanho et al., 2008; Souza et al., 2008a, 2011).

Teste é o processo de execução de um programa com a intenção de evidenciar a presença de defeitos (Myers, 2004). Para tais evidências serem encontrados, existe uma série de atividades, coletivamente chamada de Validação, Verificação e Teste, ou VV&T, com a finalidade de garantir que tanto o modo pelo qual o software está sendo construído quanto o produto em si estejam em conformidade com o especificado (Delamaro et al., 2007b). Essas atividades não se restringem ao produto final, pelo contrário, devem ser conduzidas durante todo o processo de desenvolvimento de software. As atividades de VV&T são divididas em estáticas e dinâmicas. As atividades estáticas são aquelas que não necessitam a execução ou mesmo a existência de um programa ou modelo executável para serem conduzidas. Já as dinâmicas, necessitam de um programa ou um modelo executável (Delamaro et al., 2007b).

A literatura estabelece vários significados específicos para este e outros termos relacionados como: falha, defeito, erro e engano. A seguir é dada uma definição mais clara de cada um destes termos seguindo a terminologia Padrão IEEE 610.12-1990 (IEEE, 1990).

Diz-se “defeito” (do inglês *fault*) quando um passo, processo ou definição de dados estão incorretos e “engano” (*mistake*) quando uma ação humana produz um defeito. Estes termos são estáticos, pois estão associados a um determinado programa ou modelo e não dependem de uma execução particular. A existência de um defeito pode ocasionar a ocorrência de um “erro”(*error*) durante uma execução do programa, que se caracteriza por um estado inconsistente ou inesperado. Quando um erro ocorre, pode acontecer de uma “falha”(*failure*) ser observada, ou seja, pode fazer com que o resultado produzido pela execução seja diferente do resultado esperado (Delamaro et al., 2007b).

Os programas são compostos por domínios de entrada e saída. O domínio de entrada de um programa $\mathbf{P}$, denotado por $D(\mathbf{P})$, é o conjunto de todos os possíveis valores que podem ser utilizados para executar $\mathbf{P}$. O domínio de saída de um programa, denotado por $S(\mathbf{P})$ é um conjunto de todos os possíveis resultados produzidos por $\mathbf{P}$.

Define-se “dado de teste” de um programa $\mathbf{P}$ como um elemento do domínio de entrada de $\mathbf{P}$. Um par formado por um dado de teste mais o resultado esperado — ou seja, um elemento do domínio de saída — para a execução é denominado “caso de teste”(Delamaro et al., 2007b). Por exemplo, em um programa que computa x^y , têm-se os seguintes casos de teste: $\langle(2, 2), 4\rangle$, $\langle(2, 3), 8\rangle$, $\langle(0, -1), \text{Erro}\rangle$. Um conjunto de casos de teste usados durante uma determinada atividade de teste costuma-se chamar “conjunto de teste” ou “conjunto de casos de teste”.

A Figura 2.1 configura um ambiente típico da atividade de teste. Ao definir um conjunto de casos de teste T , executa-se o programa em teste com T e verifica-se qual é o resultado obtido. Se o resultado produzido pela execução de P coincide com o resultado esperado, então nenhuma falha foi identificada. Do contrário uma falha foi revalada. O testador da Figura 2.1 está desempenhando um papel que comumente é chamado de “oráculo”, ou seja, aquele instrumento que decide se a saída obtida de uma determinada execução coincide com a saída esperada ou não.

O teste de software é composto por basicamente quatro etapas. São elas: 1) planejamento, na qual são definidas quais técnicas de teste utilizar e o planejamento da execução dos testes. 2) Projeto de casos de teste, na qual são definidos os casos de teste para execução do programa. 3) Execução, na qual o programa é executado com os dados de

Figura 2.1: Cenário Típico da Atividade de Teste (Delamaro et al., 2007b).

teste. 4) Análise, na qual os resultados são analisados para determinar se o programa produz uma saída esperada. Essas atividades devem ser conduzidas durante o processo de desenvolvimento de software e se encontram dispostas em três fases de teste: de unidade, de integração e do sistema (Delamaro et al., 2007b; Myers, 2004; Pressman, 2010).

O teste de unidade inicia o teste pela menor unidade do programa, uma sub-rotina, um procedimento ou função. O teste de unidade pode ser aplicado à medida que ocorre a implementação das unidades e pelo próprio programador, sem a necessidade de dispor-se do sistema totalmente finalizado.

Após as unidades serem testadas o teste de integração é realizado. Aqui é verificado se a junção entre essas unidades funciona corretamente. O objetivo é encontrar falhas de comunicação entre os módulos associados à interface de cada um em relação ao outro (Pressman, 2010).

O teste de sistema é executado após todas as partes integradas, ou seja, quando o sistema estiver completo. O objetivo é verificar se as funcionalidades especificadas no documento de requisitos estão corretamente implementadas. Nessa fase, além dos requisitos funcionais, os requisitos não funcionais, como segurança, desempenho e robustez, também são avaliados.

Além das três fases já citadas existe também outra fase chamada teste de regressão. O teste de regressão é realizado durante a manutenção do software, quando o software já está em uso pelo usuário. Toda vez que uma modificação é realizada no sistema, após sua liberação, corre-se o risco de novos defeitos serem introduzidos. Assim o teste de regressão verifica se os novos requisitos funcionam como esperado ou se as modificações efetuadas estão corretas.

Quando um software é testado o ideal seria que ele fosse testado com todo o domínio de entrada. Dessa forma, seria possível ter certeza que o programa estaria isento de defeito. Como já dito, o domínio de entrada de um programa P , denotado por $D(P)$, é o conjunto de todos os valores que podem ser utilizados para executar P . Porém não é viável testar P com todos os elementos de $D(P)$, pois isso necessitaria muito tempo e um grande processamento.

Para resolver esse problema o ideal é usar apenas um subconjunto reduzido de $D(P)$, para execução de P , mas que tenha alta probabilidade de revelar a presença de defeitos.

Um subconjunto de $D(P)$ é um subconjunto do domínio de entrada que contém dados de teste “semelhantes”(Delamaro et al., 2007b; Myers, 2004).

Existem duas formas de selecionar elementos de cada um dos subdomínios de entrada. O “teste aleatório” é uma delas. Aqui um grande número de dados de teste é selecionado aleatoriamente, de modo que, probabilisticamente se tenha uma boa chance de que todos os subdomínios estejam representados no conjunto de teste T . A outra forma é chamada “teste de partição” ou “teste de subdomínio”, no qual procura-se estabelecer quais subdomínios serão utilizados, e então seleciona-se os dados de teste em cada subdomínio. Para identificar esses subdomínios são definidas “regras” para detectar quando dados de teste devem estar no mesmo subdomínio ou não. Dependendo da regra que se utiliza obtêm-se subdomínios diferentes, que significa dizer diferentes casos de teste. Essas regras são chamadas “critérios de teste”. A seguir, as principais técnicas e critérios de teste existentes são descritos.

2.2.1 Técnicas e Critérios de Teste

Durante a seleção de dados de teste, diferentes técnicas e critérios podem ser empregados. As técnicas se diferenciam pela origem da informação a ser utilizada durante o teste e são basicamente divididas em técnica funcional, estrutural e baseada em defeitos.

Na técnica de teste funcional o testador desconhece a estrutura interna do programa. Apenas são conhecidas as entradas e são avaliadas as saídas. Essa técnica é baseada exclusivamente na especificação dos requisitos do sistema, não há nenhum conhecimento de como o programa está implementado para gerar os casos de testes.

Nesta técnica, primeiramente a especificação de requisitos é analisada, então entradas são selecionadas para determinar se o software comporta-se adequadamente. Os casos de testes são construídos de acordo com essas entradas. O conjunto de teste é executado e as saídas obtidas e esperadas são comparadas. Por fim, os resultados são analisados para verificar se falhas foram detectadas.

Por ser baseada na especificação, a qualidade dos critérios depende de uma boa especificação de requisitos. Dessa forma, pode ter limitações por conta de especificações incompletas ou ausentes que tornam difícil a aplicação desse critério e também não garante que partes críticas e essenciais do código tenham sido executadas. Uma vantagem do teste funcional é que ele pode ser aplicado em todas as fases de teste, por necessitar apenas da especificação do produto para derivar os casos de teste. Outra vantagem é que pode ser aplicado em qualquer paradigma de programação, pois não leva em consideração o código-fonte (Fabbri et al., 2007). Os critérios mais conhecidos da técnica funcional são: Particionamento em Classes de Equivalência, Análise de Valor Limite, Grafo Causa-Efeito e *Error-Guessing* (Fabbri et al., 2007).

A técnica de teste estrutural diferentemente da técnica funcional, baseia-se no conhecimento da estrutura interna do programa, possibilitando assim uma análise mais elaborada do comportamento do mesmo. Para tanto, se faz necessária a implementação de parte ou componentes do código. Essa técnica deriva os casos de teste a partir de uma análise ló-

gica do programa (Barbosa et al., 2007). Segundo Soares (2007), o teste estrutural requer do testador atenção aos pontos mais importantes ou “perigosos” do código.

Geralmente, os critérios dessa técnica utilizam uma representação do programa conhecida como Grafo de Fluxo de Controle (GFC) (Figura 2.2). Dado um programa **P** ele pode ser representado como um GFC que consiste em estabelecer uma correspondência entre vértices (nós), que representam conjuntos de instruções executadas sequencialmente, e indicar possíveis fluxos de controle entre blocos por meio das arestas (arcos), que representam uma transferência de controle entre instruções (Barbosa et al., 2007; Myers, 2004). Uma vez executado o primeiro comando do nó todos os demais são executados sequencialmente.

Figura 2.2: Exemplo do GFC (Grafo de Fluxo de Controle) para o programa Identifier (Barbosa et al., 2007)

Seja um GFC ($G = (N, E, s)$) em que N representa um conjunto de nós, e E o conjunto de arcos, e s o nó de entrada. Um caminho é uma sequência finita de nós $(n_1, n_2, \dots, n_k)$, $k \geq 2$, tal que existe um arco de n_i para n_{i+1} para $i = 1, 2, \dots, k - 1$. Um caminho é um caminho simples se todos os nós que compõem esse caminho, exceto possivelmente o primeiro e o último, são distintos; se todos são distintos, diz-se que esse caminho é um caminho livre de laço. Um caminho completo é aquele em que o primeiro nó é o nó de entrada e o último nó é um nó de saída do grafo G .

Os critérios estruturais aplicados em programas sequenciais geralmente são classificados em:

1. Critérios Baseados na Complexidade que utilizam informações sobre a complexidade para derivar os requisitos de teste. Um dos critérios mais conhecidos é o critério de McCabe, que se baseia na complexidade ciclomática. A complexidade ciclomática é uma métrica de software que proporciona uma medida quantitativa da complexidade

lógica do programa. O valor da complexidade ciclomática estabelece o número de caminhos linearmente independentes do conjunto básico do programa, projetando os casos de teste de tal forma que todas as arestas sejam executadas pelo menos uma vez (Barbosa et al., 2007).

2. Critérios Baseados em Fluxo de Controle que utilizam características de controle de execução do programa (Barbosa et al., 2007). Os critérios baseados em fluxo de controle mais conhecidos são: Todos-Nós, Todas-Arestas e Todos-Caminhos.
3. Critérios Baseados em Fluxo de Dados que utilizam a análise de fluxo de dados como fonte de informação para derivar os requisitos de teste. Esses critérios selecionam caminhos de teste de acordo com as localizações de definições e usos de variáveis no programa (Pressman, 2010). Esses critérios testam como os dados são utilizados nas computações estabelecendo-se associações que buscam avaliar se todas as variáveis definidas são utilizadas, como também, se todos os diferentes usos foram exercitados. Rapps e Weyuker (1982, 1985) propuseram uma família de critérios de fluxo de dados. Os principais critérios dessa família são: Todas-Definições, Todos-Usos e Todos-Du-Caminhos.

A técnica de teste baseada em defeitos utiliza o histórico dos defeitos mais comuns que ocorrem durante o processo de desenvolvimento de software para derivar os requisitos de teste. O objetivo dessa técnica é construir casos de teste que revelem a presença ou ausência de defeitos específicos (DeMillo et al., 1987).

O Teste de Mutação é um critério de teste da técnica baseada em defeitos. Neste tipo de teste o programa que está sendo testado é modificado várias vezes, criando-se um conjunto de programas alternativos ou mutantes. Após essa tarefa, o testador escolhe casos de teste que mostrem a diferença de comportamento entre o programa original e os programas mutantes.

Para geração dos mutantes são utilizados os operadores de mutação. O operador de mutação aplicado ao programa P transforma P em um programa similar (mutante). Cada mutante possui todos os comandos de P exceto o comando retirado ou alterado pelo operador de mutação. Na execução do programa em teste P com os casos de testes, verifica-se se o comportamento foi conforme o esperado. Caso não seja, então o mesmo contém defeitos e devem ser corridos.

As limitações no teste de mutação é que o custo da aplicação é elevado, pois podem ser gerados muitos mutantes, dos quais alguns podem ser equivalentes (ou seja, quando o programa mutante e o original apresentam o mesmo resultado, para qualquer caso de teste pertencente ao domínio de entrada). O número de mutantes equivalentes pode ser alto, dificultando a análise devido ao problema de mutantes equivalentes ser indecidível. A vantagem desse tipo de teste é que ele é altamente eficaz em revelar a presença de defeitos. Como não requer a identificação de estruturas como fluxo de dados e fluxo de controle pode ser aplicado em outros contextos, como exemplo em máquinas de estado finito (Delamaro et al., 2007a).

2.3 Programação Concorrente

Um programa concorrente, segundo Andrews (2000), é formado por dois ou mais processos ou *threads* que trabalham juntos para realizar uma tarefa. Seu surgimento aconteceu na década de 1960 em sistemas operacionais multiprogramados com o objetivo de diminuir ciclos ociosos do processador. Esse paradigma de programação permite o desenvolvimento de algoritmos que quando executados originam novos processos que podem executar concorrentemente e que interagem entre si. Sua aplicação abrange várias áreas como: financeira, científica, comerciais, redes de sensores, computação ubíqua entre outras.

Atualmente, o uso de programação concorrente vem ganhando bastante notoriedade devido aos atuais sistemas distribuídos e a computação de alto desempenho, os quais buscam nesse paradigma soluções para aumentar o desempenho e melhorar uso de recursos distribuídos. A programação concorrente possui várias características que a diferenciam da programação sequencial como: visão no mínimo bidimensional da aplicação, comunicação paralela, identificação do paralelismo, uso de comunicação e sincronização, gerência de dados locais e compartilhados, sincronização de diferentes estágios da execução entre outras.

A proposta da programação concorrente é que uma determinada aplicação seja dividida em vários processos concorrentes, em que cada processo é incumbido de resolver o problema em questão. Essa divisão em processos é obtida por meio de recursos de software adicionais que são responsáveis pela ativação e finalização de processos concorrentes.

Segundo Tanenbaum (2007b), um processo é um programa em execução, incluindo os valores atuais do contador de programa, registradores e variáveis e *thread* é uma linha de controle independente dentro do processo.

Para cada paradigma de programação concorrente existem formas particulares de realizar a comunicação e sincronização. Se a programação segue o paradigma de passagem de mensagem os programas normalmente são compostos por processos; ambientes de memória compartilhada são compostos por *threads* (Tanenbaum, 2007b). Vale ressaltar também que em alguns casos a aplicação concorrente pode envolver ambos os paradigmas para a sua solução. Esses paradigmas são descritos a seguir.

2.3.1 Paradigma de Memória Compartilhada

Muitas empresas de hardware e software desenvolvem sua própria API que implementa o paradigma de memória compartilhada para uso de *threads* em programação concorrente. Algumas dessas APIs, são: NT *threads*, Solaris, Java *threads* e P*Threads*.

No paradigma de memória compartilhada as *threads* utilizam variáveis compartilhadas para interagirem entre si, permitindo assim a comunicação e sincronização. A comunicação ocorre de forma implícita, bastando para isso que a *thread* transmissora escreva o dado na variável compartilhada e que a *thread* receptora leia o dado dessa variável.

O acesso exclusivo às variáveis compartilhadas é importante para a comunicação, principalmente quando existe escrita nessas variáveis. Garantindo a exclusividade ao acesso evita-se que dados sejam corrompidos devido à sobreposição da escrita e/ou utilização de dados antigos ocasionados por *threads* simultâneas. Para solucionar esse problema é usada a sincronização de controle de acesso.

O surgimento dos mecanismos de sincronização deu-se devido à evolução dos sistemas operacionais que passaram a ter o conceito de multiprogramação. As sincronizações em sistemas operacionais tinham por principal objetivo a implementação de exclusão mútua para proteger dados quando ocorresse preempção de processos (Tanenbaum, 2007a). Todas as soluções apresentadas para realizar a exclusão mútua, até então, eram ineficientes, pois apresentavam outro problema chamado espera ocupada. Esse problema, que é execução indefinida de um laço que testa a possibilidade de acesso à região crítica, é indesejado, pois desperdiça ciclos de CPU. Para resolver este problema foram criadas duas primitivas: *Sleep* e *Wakeup*. A primitiva *Sleep* bloqueia o processo até que outro chame a primitiva *Wakeup* (Tanenbaum e Steen, 2006).

Um programa concorrente com memória compartilhada somente comunicará corretamente se as sincronizações estiverem corretas. Para garantir o sucesso de uma sincronização vários mecanismos foram desenvolvidos, como semáforos e monitores, que permitem garantir a ordem de execução das *threads* e também a ordem de leitura/escrita das variáveis compartilhadas.

Semáforo é um mecanismo de sincronização, composto por uma variável inteira normalmente iniciada com 1 e uma lista de processos inicialmente vazia. O semáforo admite duas operações atômicas: **P**, comumente referenciada como *down* e **V**, comumente referenciada como *up*. O semáforo é operado da seguinte forma: a operação *down* verifica se valor da variável é maior que 0. Nesse caso o valor é decrementado e o processo segue sua execução. Caso o valor seja 0, o processo é bloqueado e colocado para dormir em uma lista de espera. A operação *up* sobre o semáforo incrementa uma unidade no seu valor e libera um dos processos da lista de espera (caso exista algum), este escolhido pelo sistema operacional. Caso uma ou mais operações *up* sejam executadas antes de uma operação *down*, não haverá a perda dessa chamada. Isso é viabilizado pela variável associada ao semáforo, a qual é incrementada a cada operação *up* (Tanenbaum, 2007b).

Um caso especial de semáforo é o semáforo binário, que também pode assumir dois valores 0 e 1. Esse tipo de semáforo também conhecido como *mutex* (**MUT**ual **EX**clusion) recebe esse nome porque tem como objetivo garantir a exclusão mútua no acesso à região crítica, que significa que apesar de qualquer processo, em qualquer momento, pode solicitar operações do monitor, apenas um processo terá permissão de execução em um determinado momento (Tanenbaum, 2007b).

Os monitores são mecanismos de sincronização abstratos, semelhantes a um pacote contendo procedimentos, variáveis e estruturas de dados de alto nível. Esse mecanismo de sincronização adiciona vários detalhes de implementação existentes nos semáforos. Sua utilização requer a definição de um novo tipo de variável, chamada variável de condição, para tornar possível a sincronização de controle de sequência que admite duas operações: *Wait* e *Signal*. Essas duas operações são semelhantes às funções *Sleep* e *Wakeup*,

respectivamente, sendo associadas a uma condição da lógica do programa (Tanenbaum, 2007b).

Existem outros mecanismos de mais alto nível para fazer a sincronização de *threads* como barreira e semáforos leitores e escritores, que geralmente são baseados em semáforos e monitores (Dongarra et al., 2003; Grama et al., 2003; Lewis e Berg, 1996).

2.3.2 Passagem de Mensagem

Quando a passagem de mensagens é usada para comunicação e sincronização, os processos enviam e recebem informações em vez de compartilhar variáveis. A troca de mensagens entre dois processos — comunicação ponto-a-ponto — é feita basicamente por meio de primitivas *send/receive*. O mecanismo de comunicação ponto-a-ponto permite que dois processos concorrentes realizem uma comunicação unidirecional (Almasi e Gottlieb, 1994).

As primitivas *send/receive* podem apresentar formas diferentes em sua semântica dependendo da sua implementação. Essas formas estão relacionadas principalmente aos conceitos bloqueante e não bloqueante, síncrono e assíncrono e se existem ou não *buffers* intermediários para armazenar temporariamente a mensagem (Dongarra et al., 2003; Grama et al., 2003).

Primitivas *send/receive* bloqueantes são liberadas da execução (permite a execução da próxima instrução) apenas quando a mensagem está segura. Isso significa que a mensagem foi efetivamente enviada, no caso de um *send* sem *buffers*, ou que a mensagem foi copiada para um *buffer* intermediário, no caso de um *send* com *buffer*. O *receive* bloqueante só é liberado quando a mensagem é recebida, ou do *buffer* ou diretamente do *send* (caso não haja *buffers*) (Dongarra et al., 2003; Grama et al., 2003).

Já as primitivas *send/receive* não bloqueantes não esperam que a mensagem esteja segura, ou seja, que tenha sido copiada para um *buffer* ou que tenha sido de fato enviada para o *receive*. Fica a critério do desenvolvedor decidir não alterar o conteúdo da mensagem na memória local até que a mesma esteja segura (Grama et al., 2003).

As primitivas *send/receive* síncronas, no nível de aplicação, obrigam a existência de um sincronismo entre a execução do *send* e do *receive*. Em contrapartida, as primitivas assíncronas não garantem esse sincronismo durante a execução.

Há também primitivas alternativas que são abstrações construídas sobre as primitivas *send/receive*. Por exemplo: chamadas de procedimentos remotos (RPC – *Remote Procedure Calls*), Invocação Remota de Métodos (RMI – *Remote Method Invocation*) e primitivas coletivas de comunicação — geralmente envolvendo mais que dois processos (Almasi e Gottlieb, 1994; Hwang e Xu, 1998; Quinn, 2004).

As API's PVM (*Parallel Virtual Machine*) e MPI (*Message Passing Interface*) oferecem infraestruturas que implementam o paradigma de passagem de mensagem. A seguir o padrão MPI é apresentado com mais detalhes pois está dentro do escopo deste trabalho.

O Padrão MPI

O padrão MPI surgiu com o objetivo de padronização dos diversos ambientes com passagem de mensagem por meio de uma especificação para o desenvolvimento de aplicações

paralelas. Ele especifica uma biblioteca sintática e semântica de funções paralelas para as linguagens sequenciais C, Fortran e C++ (Pacheco, 1996). 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.

O padrão MPI permite duas formas de comunicação entre os processos: *one-sided* e *two-sided*. Na comunicação *one-sided* o processo acessa o endereço de memória de outro processo sem que tenha participação explícita na operação. Já na comunicação *two-sided* é necessário que os dois processos tenham participado da operação. Essa comunicação pode ocorrer tanto com a utilização ponto-a-ponto quanto à utilização de funções, que são descritas a seguir.

Em MPI uma computação é formada por um ou mais processos que se comunicam por meio de chamadas de sub-rotinas de envio e recebimento de mensagens. Essa comunicação ocorre de duas formas: a primeira utiliza primitivas *send* e *receive*, que é conhecida como comunicação ponto-a-ponto; a segunda utiliza funções como *broadcast*, *barrier*, *gather*, entre outras, é conhecida como comunicação coletiva.

Uma função em MPI é uma função em C ou procedimento em Fortran que realiza uma determinada atividade. Quando se necessita utilizar uma função MPI, deve-se sinalizar que uma determinada parte do sistema será executada em paralelo (Quinn, 2004). Existem várias funções disponíveis em MPI para diversas finalidades. A seguir são apresentadas e detalhadas algumas funções disponíveis em MPI.

A função *MPLInit* é a primeira função que todo processo MPI deve chamar. Essa função permite que o sistema possa acessar a biblioteca MPI quando necessário ao decorrer da execução do programa.

Quando o programa MPI é inicializado cada processo ativo torna-se membro de um comunicador chamado *MPLCOMM_WORLD*. Esse comunicador é responsável por criar um ambiente para a passagem de mensagem entre os processos. Os processos presentes no comunicador são ordenados por um *rank* único que pode estar entre 0 e $p - 1$, onde p é o número de processos. O processo chama a função *MPLCOMM_rank* para determinar seu *rank*, que esta por sua vez, pode ser usada para determinar qual parte da tarefa o processo é responsável. O processo também chama a função *MPLCOMM_size* para determinar o número total de processos no comunicador.

As funções *MPLSend* e *MPLRecv* permitem a comunicação de envio e recepção entre processos respetivamente. Em MPI, o envio de mensagem ponto-a-ponto entre os processos pode ser implementado de quatro modos diferentes:

Standard - no qual a implementação decidirá a existência de um *buffer* para guardar a mensagem. Este modo é dito como não local, pois a operação de envio pode depender da postagem de um *receive* correspondente para completar.

Ready - no qual o envio da mensagem só é iniciado se um *receive* correspondente já tenha sido postado no processo destino, caso contrário um erro é gerado.

Synchronous - no qual a operação de envio de mensagem completa quando um *receive* correspondente tiver sido postado. Este modo é dito como não local, pois a operação de envio depende da postagem de *receive* correspondente para completar.

Buffered - no qual a operação de envio de mensagem completa independentemente da postagem de um *receive* correspondente. Este modo é dito local, pois somente a execução do processo local é suficiente para que a operação seja completada.

Além dos modos de comunicação, as operações de *send* e *receive* podem ser bloqueantes e não bloqueantes com a finalidade de melhorar o desempenho e evitar *deadlocks*. Nas operações bloqueantes, a função chamada não pode retornar até que os dados da mensagem tenham sido armazenados de maneira que o *buffer* de comunicação possa ser sobrescrito. Já nas operações não bloqueantes, uma função pode retornar antes de completar, tornando necessário chamar outra função para verificar se a operação foi completada.

Ainda com relação às funções de envio e recebimento de mensagens, o padrão MPI oferece a função *MPLSendrecv* que realiza o envio e o recebimento de uma mensagem em apenas uma chamada de função.

As funções *MPLStart* e *MPLStartall* são utilizadas para comunicação persistente, nas quais as funções de envio e/ou recebimento são definidas previamente. A função *MPLStart* realiza a inicialização de comunicação persistente usando uma requisição como argumento. A função *MPLStartall* realiza a inicialização de um conjunto comunicação persistente usando um conjunto de requisições como argumento.

Além das funções ponto-a-ponto, o MPI também possui funções coletivas, as quais envolvem três ou mais processos. Uma dessas funções é a função *MPLReduce* que realiza uma determinada operação aritmética sobre os valores de todos os processos dentro de um determinado grupo e envia o resultado a um determinado processo. A função *MPLAllreduce* também realiza uma operação aritmética sobre os valores de todos os processos dentro de um determinado grupo gerando um vetor de resultados, o qual será enviado a um determinado processo.

O padrão MPI oferece funções que possibilitam a espera até que uma requisição seja completada. A função *MPLWait* tem como objetivo esperar até que uma requisição seja completa. A função *MPLWaitall* é similar a função *MPLWait*, porém trabalha com um conjunto de requisições e não somente uma. A função *MPLWaitany* tem como objetivo bloquear até que uma das operações associadas às requisições presentes no *array* seja completada.

Existem duas funções que são utilizadas para finalizar processos em MPI: *MPLAbort* e *MPLFinalize*. A função *MPLAbort* realiza o término dos processos pertencentes a um determinado comunicador. A função *MPLFinalize* realiza a finalização do ambiente MPI e permite que o sistema libere os recursos, como por exemplo memória, que foram alocados para o MPI. Essa função deve ser executada pelo processo que iniciou o ambiente MPI, ou seja, que chamou a função *MPLInit*.

Na Figura 2.3 é apresentado um exemplo de programa concorrente, chamado GCD (*Greatest Common Divisor*), implementado em MPI. Esse programa implementa um algoritmo para o cálculo do máximo divisor comum entre três números inteiros. No código apresentado é possível observar as funções de inicialização (linha 19) e finalização (linha 32) do código paralelo. Pode-se também observar as funções de envio de mensagens (linha 42) e recebimento (linha 46). A função utilizada para determinar o *rank* do processo (linha 20).

No caso do programa GCD, os argumentos passados por linha de comando são definidos da seguinte forma : `mpirun -np 4 gcd 10 30 30`, onde “`mpirun -np`”, é um comando para executar um program MPI, “4” é a quantidade de processos concorrentes que serão criados, “gcd” é o nome do executável e “10 30 30” são as entradas para executar o programa. A função `main` verifica se a quantidade de processos passados por argumentos é igual a 4. Caso seja verdadeiro, inicia-se o código concorrente com a declaração da função `MPLInit`. Em seguida o processo determina seu `rank` dentro do comunicador utilizando a função `MPLComm_rank`. O processo de `rank` é 0 ficará responsável por executar o código do processo Mestre e os demais executarão o código correspondente ao processo Escravo. O processo Mestre pega os dois primeiros números e envia para o processo 1 (linha 42) e os dois últimos para o processo 2 (linha 45) usando a função `MPLSend`.

Os dois processos Escravos recebem os valores enviados pelo processo Mestre, através da função `MPLRecv`, encontram o máximo divisor comum entre os dois valores, utilizando a técnica de sucessivas subtrações, e enviam o resultado para o processo Mestre (linha 80). Se um dos valores retornados ao Mestre for 1, então o Escravo 3 não recebe nada e é finalizado. Se esses valores possuem valores diferentes de 1, então esse par de valores é enviado ao Escravo 3 (linha 54) para que ele determine o GCD dos mesmos. O resultado é retornado ao processo Mestre que imprime o resultado final (linha 66).

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4 #include "gcd.h"
5
6 void Master(int, int, int);
7 void Slave(int rank);
8
9 int main(int argc, char** argv)
10 {
11     int myRank;
12     int x, y, z;
13
14     if (argc != 4)
15     {
16         puts("argc != 4");
17         return 1;
18     }
19     MPI_Init(&argc, &argv);
20     MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
21     if (myRank == 0)
22     {
23         x = atoi(argv[1]);
24         y = atoi(argv[2]);
25         z = atoi(argv[3]);
26         Master(x, y, z);
27     }
28     else
29     {
30         Slave(myRank);
31     }
32     MPI_Finalize();
33     return 0;
34 }
35
36 //Processo Mestre
37 void Master(int x, int y, int z)
38 {
39     int buf[3];
40     buf[0] = x;
41     buf[1] = y;
42     MPI_Send(buf, 2, MPI_INT, 1, 1, MPI_COMM_WORLD);
43     buf[0] = y;
44     buf[1] = z;
45     MPI_Send(buf, 2, MPI_INT, 2, 1, MPI_COMM_WORLD);
46     MPI_Recv(buf, 1, MPI_INT, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &status);
47     x = buf[0];
48     MPI_Recv(buf, 1, MPI_INT, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &status);
49     y = buf[0];
50     if (x > 1 && y > 1)
51     {
52         buf[0] = x;
53         buf[1] = y;
54         MPI_Send(buf, 2, MPI_INT, 3, 1, MPI_COMM_WORLD);
55         MPI_Recv(buf, 1, MPI_INT, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &status);
56         z = buf[0];
57     }
58     else
59     {
60         buf[0] = -1;
61         buf[1] = -1;
62         MPI_Send(buf, 2, MPI_INT, 3, 1, MPI_COMM_WORLD);
63         MPI_Recv(buf, 1, MPI_INT, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &status);
64         z = 1;
65     }
66     printf("resultado = %d\n", z);
67 }
68 //Processo Escravo
69 void Slave(int rank)
70 {
71     int buf[2];
72     MPI_Recv(buf, 2, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
73     while (buf[0] != buf[1])
74     {
75         if (buf[0] < buf[1])
76             buf[1] = buf[1] - buf[0];
77         else
78             buf[0] = buf[0] - buf[1];
79     }
80     MPI_Send(buf, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
81 }

```

Figura 2.3: Código fonte do programa GCD em MPI

2.3.3 Teste Estrutural de Programas Concorrentes

Dada a importância da programação concorrente também é fundamental que esses programas apresentem boa qualidade e que apresentem o menor número de defeitos possível. Para isso, técnicas e critérios de teste precisam ser desenvolvidos especificamente para o universo dos programas concorrentes, uma vez que aspectos não presentes em sistemas sequenciais como comunicação, sincronização e não determinismo devem ser considerados durante a definição de mecanismos de apoio à atividade de teste.

O não determinismo permite que diferentes execuções de um programa concorrente, com o mesmo valor de entrada, possam produzir diferentes saídas corretas. Esse aspecto torna a atividade de teste em programas concorrentes mais complexa, porém é possível aproveitar e adaptar, com sucesso, critérios definidos para programas sequencias.

Para definição de critérios de teste em programas concorrentes dois pontos importantes devem ser considerados: os tipos de defeitos que devem ser evidenciados pelos critérios de teste e como representar o programa concorrente de modo a obter as informações necessárias para os critérios de teste.

Krawczyk et al. (1994) apresentam dois tipos de erros relacionados a programas concorrentes. Seja um caminho p_x , $D(p_x)$ é o seu domínio de entrada, isto é, $D(p_x)$ é formado pelas entradas que exercitam p_x e $C(p_x)$ é a computação de p_x , isto é, $C(p_x)$ é formada pelas saídas produzidas pela execução de p_x . Os dois tipos de erros definidos são:

- **erro de observabilidade** (*observability error*): ocorre em um programa P se para algum caminho $p_i \in P$ existe outro caminho $p_j \in P$ tal que $D(p_i) = D(p_j)$ e $C(p_i) \neq C(p_j)$.
- **erro de travamento** (*locking error*): ocorre em um programa P se para algum caminho $p \in P$, formado pelo conjunto de nós $p = q_0 q_1 \dots q_j$, existe um nó q_i em p tal que o subcaminho $p_s = q_0 q_1 \dots q_i$ contendo um domínio $D(p_s)$, existe pelo menos um elemento de $D(p_s)$ em que todos os predicados avaliados a partir de q_i são falsos.

O erro de observabilidade é relacionado ao ambiente de teste e ocorre quando o testador não pode controlar o comportamento do programa executado em paralelo. Já o erro de travamento ocorre, geralmente, quando há erros de comunicação entre processos.

DeSouza et al. (2005) apresentam uma taxonomia de erros para programas desenvolvidos em MPI (*Message Passing Interface*). A taxonomia apresenta três principais categorias de erros: 1) Sincronização, na qual se pode encontrar erros envolvendo *deadlock* e *data race*. 2) Inadequação, que ocorre quando, por exemplo, uma chamada é realizada de forma errada, erro no tipo dos argumentos passados ou no tamanho da mensagem. 3) Recursos, que são erros envolvendo recursos como, por exemplo, tipos de dados ou comunicadores.

Assim como nos programas sequencias, os programas concorrentes precisam ser representados de modo a obter informações necessárias para os critérios de teste. Na Seção 2.2 foi apresentado que critérios de teste estruturais de programas sequenciais utilizam

um grafo para representar o código do programa e extrair informações necessárias para aplicação desses critérios. Os modelos de teste são utilizados para representar o programa a ser testado, neles são contidas informações relevantes para o teste, como fluxo de dados, controle, caminhos etc. Os modelos de teste existentes para programas concorrentes são extensões de modelos de programas sequenciais e em sua maioria são baseados em modelos de estados ou modelos de fluxo de controle de dados (Souza et al., 2007).

O teste estrutural é uma técnica de teste amplamente utilizada em programas sequencias e foi estendida para o contexto de programas concorrentes por Taylor et al. (1992). O trabalho de Taylor et al. (1992) concerne na criação de um grafo de concorrência (ou grafo de alcançabilidade) a partir da análise estática de programas concorrentes escrito em Ada e CSP. O grafo de concorrência é criado a partir de possíveis estados de concorrência, levando em consideração a comunicação e sincronização do programa concorrente. Cada estado de concorrência no grafo é representado por um nó de onde um arco é direcionado para cada possível estado de concorrência seguinte.

Em Yang et al. (1998) os autores apresentam um modelo capaz de representar os aspectos de comunicação entre processos em um programa concorrente. Basicamente o trabalho propõe a construção de um Grafo de Fluxo de Controle (GFC) para cada processo paralelo, da mesma forma que ocorre em programas sequenciais. A partir desses grafos um Grafo de Fluxo de Controle Paralelo é construído, composto dos GFCs dos processos, adicionando arestas de sincronização. Essas arestas representam a criação dos processos paralelos e a comunicação entre esses processos.

As ideias propostas por Yang et al. (1998) foram estendidas por Souza et al. (2008b) que apresentam um conjunto de critérios para serem aplicados em programas concorrentes que utilizam paradigma de passagem de mensagem. Os autores consideram como aspectos importantes para o teste desses programas a comunicação e a sincronização. A principal diferença entre as abordagens é que Souza et al. (2008b) consideram processos paralelos executando em espaços diferentes de memória, sendo assim necessário representar a comunicação e a sincronização dos processos.

O modelo de teste proposto por Souza et al. (2008b) faz algumas considerações: 1) O programa concorrente é formado por conjunto de n processos que executam concorrentemente; 2) O número n de processos é fixo e conhecido em tempo de compilação e todos os processos são criados no momento de inicialização do programa concorrente; 3) Os processos se comunicam por meio de diferentes diretivas *send* e *receive* que podem acontecer de forma ponto-a-ponto, em que um processo envia uma mensagem para um processo específico, ou de forma coletiva, onde um processo manda uma mensagem para um grupo de outros processos. O modelo consiste em construir um PCFG (*Parallel Control Flow Graph*) que é criado a partir da união dos CFGs de cada processo que compõe o programa concorrente. Esses processos são interconectados por arestas que representam a comunicação/sincronização entre os processo.

Um nó i em um processo p é representado pela notação n_i^p . Com a construção do PCFG, pode-se identificar N que representa o conjunto de nós e E que representa o conjunto de arestas. N possui dois subconjuntos: N_s e N_r , onde N_s que representa o conjunto de nós que possuem a função *send* e N_r que representa o conjunto de nós que possuem a função *receive*. E também possui dois subconjuntos: E_i e E_s , onde E_i

representa o conjunto de arestas intraprocessos, que são arestas internas a um processo p , e E_s representa o conjunto de arestas interprocessos, que são arestas que representam a comunicação entre os processos, ou seja, arestas que ocorrem entre os processos paralelos. Para cada $n_i^p \in N_s$, há um conjunto R_i^p associado, que representa os possíveis nós que recebem a mensagem enviada pelo nó n_i^p .

Um caminho π no GFC de um processo p é chamado intraprocesso se ele não contém nenhuma aresta interprocesso. É dado por uma sequência de nós de $\pi = (n_1, n_2, \dots, n_m)$ tal que $n_i, n_{i+1} \in E_i^p$. Um caminho π no GFCP que contém pelo menos uma aresta interprocesso é dito um caminho interprocessos. Um caminho interprocessos é completo se todos os caminhos intraprocessos que o compõe também são completos.

Em programas sequencias uma variável x é definida quando um valor é armazenado na sua posição de memória. No contexto de programas concorrentes que utilizam o ambiente de passagem de mensagem, uma nova forma de definir uma variável é possível por meio da execução de uma função de comunicação como, por exemplo, o comando *receive*. O conjunto de variáveis definidas em um nó n_i^p é dado por $def(n_i^p)$.

O uso de x ocorre quando x é referenciado em uma expressão. Em programas concorrentes são caracterizados em três tipos de uso de variáveis:

- uso computacional (c-uso): ocorre em uma computação, relacionado a um nó n_i^p do GFCP;
- uso predicativo (p-uso): ocorre em uma condição (predicado), associado a comandos de fluxo de controle em arestas intraprocessos (n_i^p, n_j^p) do GFCP;
- uso comunicacional (s-uso): ocorre em comandos de sincronização (*send* e *receive*) em arestas intreprocessos $(n_i^{p1}, n_j^{p2}) \in E_s$

Como ocorre com os demais critérios de teste estrutural, um caminho $\pi = (n_1, n_2, \dots, n_j, n_k)$ é livre de definição em relação a uma variável x definida no nó n_1 e usada no nó n_k ou arestas (n_j, n_k) , se $x \in def(n_1)$ e $x \notin def(n_i)$, para $i = 2 \dots j$.

De posse dessas informações, é possível caracterizar as associações que devem ser requeridas pelos critérios estruturais aplicados a programas concorrentes. Três tipos de associações podem ser derivadas:

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

Note que as associações p-uso e c-uso são associações intraprocessos, ou seja, a definição e o uso de x ocorrem no mesmo processo p_1 e são geralmente requeridas no teste de programas sequenciais, realizando o teste de cada processo isoladamente. Uma associação s-uso pressupõe a existência de um segundo processo p_2 e pode ser considerada uma associação interprocessos. Essas associações permitem verificar erros na comunicação entre processos, ou seja, no envio e no recebimento de mensagens. Considerando associações s-usos e a comunicação entre processos, outros tipos de associações interprocessos são caracterizadas:

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

Segundo Souza et al. (2008b), as duas últimas associações foram definidas dada a motivação de que a associação c-uso e p-uso ocorrem no mesmo processo, ou seja, a definição e o uso ocorrem em nós distintos de um mesmo processo. Dessa forma, quando aplicados em programas concorrentes critérios de teste definidos para essas associações, nenhuma comunicação/sincronização é exercitada e, assim, não é possível detectar defeitos com relação à comunicação/sincronização nas associações do tipo c-uso e p-uso. É possível aplicar os critérios definidos para programas sequencias em programas concorrentes que utilizam o ambiente de passagem de mensagem, porém é necessário que isso seja feito em cada CFG separadamente. No entanto, o objetivo do modelo é testar a comunicação/sincronização dos processos, utilizando para isso o PCFG. A partir dessa consideração, os seguintes critérios de teste são propostos (Souza et al., 2008b).

Crítérios baseados em fluxo de controle e no fluxo de comunicação

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

Ainda podem-se definir critérios Todos os Caminhos do GFC_p (para $p = 0, \dots, n - 1$) e Todos os Caminhos do GFCP, equivalente a executar todas as combinações possíveis dos caminhos nos GFC_p , mas esses critérios são em geral impraticáveis.

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. Caso não exista associação s-c-uso ou s-p-uso com relação a x , é requerido qualquer outro tipo de associação interprocessos em relação a x .
- Todos-c-Usos: requer que todas as associações c-usos sejam exercitadas pelo conjunto de caso de teste.
- Todos-p-Usos: requer que todas as associações p-usos sejam exercitadas pelo conjunto de caso de teste.
- Todos-s-Usos: requer que todas as associações s-usos sejam exercitadas pelo conjunto de caso de teste.
- Todos-s-c-Usos: requer que todas as associações s-c-usos sejam exercitadas pelo conjunto de caso de teste.
- Todos-s-p-Usos: requer que todas as associações s-p-usos sejam exercitadas pelo conjunto de caso de teste.

No trabalho de Lei e Carver (2006) uma técnica chamada teste de alcançabilidade é apresentada. Nesta técnica, primeiramente é realizada uma execução não controlada (sem interferência no não determinismo) do programa paralelo gerando uma sequência de sincronização Q_0 que é gravada. Essa sequência de sincronização é gravada durante a execução por meio de *timestamp*. A partir do trace de execução de Q_0 , identificam-se condições de disputa nos eventos *receive*, gerando variantes da sequência de sincronização executada. Em seguida, essas variantes são utilizadas para realizar execuções controladas baseadas em prefixo com a mesma entrada, exercitando as sincronizações da variante. Como as sequências são derivadas a partir de uma execução do programa, essa abordagem não apresenta o problema da não executabilidade. Diferentemente das técnicas anteriores, essa técnica permite, em tempo e execução e sem a necessidade de uma análise estática anterior, identificar possíveis disputa entre as mensagens trocadas entre processos (não determinismo) e, assim, explorar todas as possibilidades de execuções que podem ser realizadas a partir de uma execução inicial. A desvantagem desse modelo é a mesma encontrada no grafo de concorrência de Taylor et al. (1992) que é a explosão de estados de concorrência uma vez que o número de estados cresce exponencialmente com o número de *threads* e o número de sincronizações executadas.

Com base nos conceitos de teste de alcançabilidade, (Souza et al., 2011) apresentam uma proposta de utilização dessa abordagem em conjunto com os critérios estruturais para programas com passagem de mensagens. Essa proposta apresenta a vantagem de reduzir o custo do teste de alcançabilidade e também guiar a avaliação dos critérios de cobertura.

2.3.3.1 Ferramenta de Teste ValiMPI

Analogamente aos programas sequenciais, o teste de programas concorrentes requer ferramentas para automatizar a fase de teste. Isso ocorre porque esse tipo de atividade realiza procedimentos complexos que são muitos custosos e, em geral, impraticáveis de serem feitos manualmente.

Existem algumas ferramentas que apoiam o teste de programas concorrente, considerando os diferentes paradigmas e abordagens de teste. Neste mestrado, foi considerada a ferramenta ValiMPI (Hausen et al., 2007), que apoia o teste estrutural de programas concorrentes em MPI e implementa os critérios de teste definidos por Souza et al. (2008b).

A ferramenta **ValiMPI** de um modo geral, permite ao usuário criar uma sessão de teste, salvar, executar dados de teste e avaliar a cobertura do teste em relação aos critérios de teste estruturais definidos em Vergilio et al. (2005). Atualmente os critérios disponíveis na ValiMPI são: Todos-Nós, Todos-Nós-r, Todos-Nós-s, Todas-Arestas, Todas-Arestas-s, Todos-c-Usos, Todos-p-Usos e Todos-s-Usos.

Figura 2.4: Arquitetura da ferramenta de teste ValiMPI (Souza et al., 2008b).

A arquitetura (veja Figura 2.4) da ValiMPI é composta por cinco módulos: ValiInst, ValiElem, ValiExec, ValiEval e ValiSync.

O módulo **ValiInst** é responsável por executar a análise estática do código fonte do programa concorrente e assim extrair informações de fluxo de controle e fluxo de dados para gerar o GFC (Grafo de Fluxo de Controle) e também pela instrumentação do programa original. Esse programa modificado é instrumentado pela IDeL (Simao et al., 2003), recebe o programa MPI e insere no código instruções para gerar traces (ou registros) da execução diferindo-o do programa original, como exemplos, inserção de *check-points*, substituição de funções `MPL_Isend`, `MPL_Recv` e `MPL_Irecv` por `ValiMPL_Send`, `ValiMPL_Isend`, `ValiMPL_Recv` e `ValiMPL_Irecv`.

O módulo **ValiElem** é responsável pela geração dos elementos requeridos para os critérios de cobertura implementados pela ValiMPI. Os elementos requeridos representam informações que precisam ser executadas (exercitadas/testadas) no código do programa concorrente. O módulo ValiElem recebe com entrada os grafos GFCs e informações sobre fluxo de dados gerando como saída, os elementos requeridos. Nesse módulo também são gerados os descritores dos elementos requeridos. Para esse propósito, dois outros grafos são usados: o grafo reduzido de herdeiros (Chusho, 1987) e o grafo(*i*). Esses grafos foram definidos com base na ferramenta de teste PokeTool (Chaim, 1991).

O módulo **ValiExec** executa o programa instrumentado com o caso de teste fornecido pelo usuário. O **ValiExec** armazena a entrada e a saída do caso de teste, os parâmetros de execução e os respectivos *traces* (registros) de execução. O *trace* de execução inclui o caminho executado por cada processo paralelo, bem como as sequências de sincronização. Ele será usado pelo módulo ValiEval para determinar quais elementos foram cobertos. O módulo **ValiExec** também possibilita a execução controlada do programa paralelo em teste. Este recurso é útil para repetir a execução de um caso de teste, garantindo que a mesma sequência de sincronizações seja exercitada. Para isso funções foram implementadas as quais garantem que as sequências de sincronizações se repitam para o mesmo caso de teste.

O módulo **ValiSync** implementa o teste de alcançabilidade proposto por Lei e Carver (2006). Com base no *traces* de execução e arquivos de sincronizações obtidos pelo módulo **ValiExec**, o módulo **ValiSync** identifica possíveis variantes de execuções do dado de teste executado tomando como base o algoritmo proposto por Lei e Carver (2006). Com a geração de *traces* parciais, a execução controlada dessas possíveis variantes pode ser realizada. Esses *traces* parciais contém as sincronizações que devem ser executadas deterministicamente. As demais sincronizações são executadas não deterministicamente.

O módulo **ValiEval** avalia a cobertura obtida pelo caso de teste em relação a um dado critério. Esse módulo usa os descritores e os elementos requeridos gerados pelo ValiElem e os caminhos executados (*traces*) pelos casos de teste para verificar quais elementos requeridos foram exercitados. O módulo implementa um autômato associado aos descritores. Então, um elemento requerido está coberto se um caminho π executado é reconhecido pelo seu autômato correspondente. A taxa de cobertura (porcentagem de elementos requeridos cobertos) e uma lista de elementos requeridos não cobertos em relação ao critério de teste selecionado são mostrados ao fim da execução desse módulo.

2.4 Técnicas para Geração de Dados de Teste

Uma das etapas mais difíceis de serem automatizadas na fase de teste é a geração de dados, pois se trata de um problema indecidível, em razão da complexidade e tamanho dos programas. A indecidibilidade advém de algumas restrições, inerentes à atividade de teste, que impossibilitam a automatização completa da fase de geração de dados (Vergilio et al., 2007).

Correção coincidente, caminho ausente, caminho não executável e mutantes equivalentes são exemplos dessas restrições. A correção coincidente ocorre quando o programa em teste possui um defeito que é alcançado por um dado de teste, um estado de erro é produzido, mas coincidentemente um resultado correto é obtido. O caminho ausente corresponde a uma determinada funcionalidade requerida para o programa, mas por engano não foi implementada, isto é, o caminho correspondente não existe no programa. Caminhos não-executáveis ocorre quando não existe um conjunto de valores atribuídos às variáveis de entrada do programa, parâmetros e variáveis globais que causam sua execução. O problema de mutantes equivalentes ocorre quando uma mudança feita no programa original para criar um mutante não altera a função implementada, ou seja, o programa mutante apresenta sempre, para qualquer valor de entrada, o mesmo comportamento do programa original.

Apesar dessas limitações, segundo Vergilio et al. (2007) encontram-se na literatura vários trabalhos que propõem técnicas que podem ser utilizadas para gerar dados de teste para satisfazer os diversos critérios de teste. Dentre as técnicas para geração de dados as mais conhecidas são: geração aleatória (Chen, 2008; Duran e Ntafos, 1984), geração com execução simbólica (Boyer et al., 1975; Offutt e Seaman, 1990), geração com execução dinâmica (Korel, 1990) e geração usando técnicas da computação evolutiva (Ghiduk et al., 2007; Zhang e Gong, 2010). Esta seção aborda de uma forma geral as diversas técnicas para geração de dados e é baseada parcialmente em McMinn (2004).

2.4.1 Geração Aleatória

A geração de dados de teste aleatória tem como objetivo selecionar pontos específicos do domínio de entrada de forma aleatória até que o critério seja satisfeito (Chen, 2008; Duran e Ntafos, 1984). É de fácil implementação, de baixo custo e produz dados próximos aos utilizados operacionalmente.

Uma vez que a geração de dados de teste busca escolher os melhores pontos do domínio de entrada, a geração de dados de teste aleatória seleciona pontos do domínio de entrada aleatoriamente até que o critério de teste seja satisfeito. Ou seja, esta técnica não garante que trechos importantes do código do programa em teste serão executados. O ideal seria que os dados de teste fossem capazes de testar pontos específicos do programa, aspecto que não é garantido com essa técnica.

2.4.2 Geração com Execução Simbólica

Geração de dados de teste com execução simbólica geralmente é usada em conjunto com a técnica baseada em caminhos. Ela tem como objetivo gerar dados de teste automáticos para um dado caminho ou um grupo de caminhos. Para isso, expressões algébricas, que representam a execução de um determinado caminho, são derivadas e o resultado é uma expressão simbólica, chamada computação do caminho que expressa as variáveis de saída em termos das variáveis de entrada, e outra expressão chamada condição de caminhos que representa condições para o caminho a ser executado (Boyer et al., 1975; Offutt e Seaman, 1990).

A aplicação dessa técnica é bastante dispendiosa, devido às derivações sistemáticas e a manipulação das expressões simbólicas. À medida que a complexidade do software aumenta, o uso dessa técnica pode se tornar proibitiva. Outra desvantagem é que a abordagem apresenta problemas para software com variáveis que dependem da execução de laços de repetição, variáveis dependentes de vetores, referência a ponteiro e chamada de métodos.

2.4.3 Geração com Execução Dinâmica

A geração de dados de teste com execução dinâmica se baseia na execução real do programa em teste, em métodos de minimização de funções e análise de fluxo de dados. Partes do programa são tratadas como funções que são avaliadas durante a execução, e os dados são avaliados e melhorados a medida que são gerados. A análise de fluxo de dados dinâmica é usada para determinar quais variáveis de entrada são responsáveis pelo comportamento incorreto do programa. O método de minimização de função é usado no processo de busca do dado mais adequado (Dara et al., 2009; Korel, 1990).

A ideia central dessa técnica é coletar dados, em tempo de execução, que se aproximam da cobertura do critério. Depois de coletados, os melhores dados são alterados para que um deles satisfaça o critério.

A proposta da abordagem dinâmica é mais genérica e resolve muitos dos problemas da abordagem com execução simbólica. Porém, muitos métodos de minimização são bastante simples e boa parte destes apresentam problemas quando a função objetivo possui mínimos e/ou máximo locais — pontos onde a função alcança seu menor e maior valores respectivamente (Michael et al., 2001).

2.4.3.1 Geração com Técnicas de Computação Evolutiva

Os Algoritmos Evolucionários usam a evolução simulada como uma estratégia de busca para desenvolver soluções candidatas usando operadores inspirados pela genética e seleção natural. Uma meta-heurística é um método heurístico usado para encontrar soluções para problemas de otimização a um custo de computação razoável. A heurística é parte de um algoritmo de otimização que utiliza informações coletadas durante a execução deste para auxiliar na decisão de qual solução candidata ao problema deve ser testada ou como a solução seguinte deve ser produzida (Michalewicz e Fogel, 2004; Rayward-Smith, 1996).

Geralmente, meta-heurística é aplicada a problemas em que se desconhece um algoritmo eficiente para sua resolução. Esses problemas podem ser conhecidos como NP-Completo ou NP-Difícil ou ainda problemas que se conhece um algoritmo capaz de resolvê-lo, em um tempo polinomial, porém é impraticável. Meta-heurística são estratégias que se adaptam a problemas específicos.

O uso de meta-heurística para geração automática de dados de teste é uma área de grande interesse entre pesquisadores. Isso ocorre principalmente pelo fato de que a geração manual de dados de teste é uma tarefa cara, dificultosa e demorada. Outra justificativa para o uso de meta-heurística é que essa técnica é uma ótima opção para lidar com o problema de indecidibilidade inerente à geração de dados de teste (McMinn, 2004).

No contexto de geração de dados de teste, meta-heurísticas são utilizadas para a transformação de um critério de teste em uma função objetivo. Essa função objetivo compara e contrasta soluções da busca em relação ao objetivo geral da mesma. Essa informação é usada para direcionar a pesquisa em locais em que haja áreas promissoras no espaço de busca (McMinn, 2004). A seguir são apresentadas algumas meta-heurísticas que são usadas para geração de dados.

2.4.3.2 Subida da Encosta (Hill Climbing)

Subida da encosta (do inglês *Hill Climbing*) é um algoritmo de busca local que é baseado na ideia de subida de um monte, no qual o objetivo é chegar ao pico mais alto (solução ótima) apenas examinando os locais mais próximos (vizinhos) (Russell e Norvig, 2009). Inicialmente, uma solução inicial é escolhida aleatoriamente do espaço de busca para então ser melhorada. Em seguida os vizinhos desta solução são analisados e caso exista alguma que possua solução melhor a solução atual é substituída. Este processo continua até que não exista mais nenhum vizinho com valor mais alto que a solução atual.

Segundo Russell e Norvig (2009) o algoritmo da Subida da Encosta nada mais é do que um laço repetitivo que se movimenta de forma contínua no sentido do valor crescente, isto é, encosta acima, e termina quando encontra o pico mais alto. A Figura 2.5 apresenta o algoritmo dessa.

```

função SUBIDA-DE-ENCOSTA(problema) retorna um estado que é um máximo local
  entradas: problema, um problema
  variáveis locais: corrente, um nó
                     vizinho, um nó

  corrente ← CRIAR-NÓ(ESTADO-INICIAL[problema])
  repita
    vizinho ← um sucessor de corrente com valor mais alto
    se VALOR[vizinho] ≤ VALOR[corrente] então retornar ESTADO[corrente]
    corrente ← vizinho

```

Figura 2.5: Algoritmo de busca de Subida da Encosta (Russell e Norvig, 2009).

Este algoritmo de busca é simples e proporciona resultados rápidos. Contudo, o fato de ir para o próximo vizinho sem decidir com antecedência o caminho que irá percorrer em seguida, pode acarretar em alguns problemas como, cair em máximos locais, picos

e platôs. O maior problema do algoritmo está relacionado à convergência prematura, principalmente quando a busca encontra um máximo local, pois se trata de um pico mais alto que cada um de seus estados vizinhos, embora seja mais baixo que o máximo global o algoritmo fica paralisado quando alcança esse estado. Os picos resultam em uma sequência de máximos locais, ocasionando uma difícil movimentação e por fim, um platô é uma região onde a função de avaliação é plana.

Existem alguns métodos para resolver o problema quando o algoritmo alcança um ponto em que não há progressos, como os reinícios aleatórios ou movimentos laterais se considerado que a natureza do problema seja uma planície. Miller e Spooner (1976) apresentam uma abordagem para geração de dados de teste que utiliza busca local.

2.4.3.3 Têmpera Simulada (Simulated Annealing)

Têmpera Simulada (do inglês *Simulated Annealing*) também conhecida com Recozimento Simulado, é uma técnica de busca local probabilística proposta inicialmente por Kirkpatrick et al. (1983) que é baseada em uma analogia à termodinâmica ao simular o arrefecimento de um metal — operação conhecida como “recozimento”, onde o metal é aquecido até sua temperatura de fusão e em seguida é realizado o processo reverso de esfriar lentamente.

Essa técnica inicia sua busca a partir de uma solução aleatória, em seguida essa solução é substituída por outra próxima que é selecionada de acordo com uma função objetivo e com uma variável T . Quanto maior for o valor de T , maior é o componente aleatório que será incluído na próxima solução selecionada. Conforme a evolução da busca, o valor de T é decrementado, levando a uma solução ótima. Na Figura 2.6 é mostrado um algoritmo de Têmpera Simulada.

```

função TEMPERA-SIMULADA (problema, escalonamento) retorna um estado solução
  entradas: problema, um problema
             escalonamento, um mapeamento de tempo para “temperatura”
  variáveis locais: corrente, um nó
                     próximo, um nó
                      $T$ , uma “temperatura” que controla a probabilidade de passos descendentes

  corrente  $\leftarrow$  CRIAR-NÓ(ESTADO-INICIAL[problema])
  para  $t \leftarrow$  até  $\infty$  faça
     $T \leftarrow$  escalonamento [ $t$ ]
    se  $T = 0$  então retornar corrente
    próximo  $\leftarrow$  um sucessor de corrente selecionado ao acaso
     $\Delta E \leftarrow$  VALOR [próximo] - VALOR [corrente]
    se  $\Delta E > 0$  então corrente  $\leftarrow$  próximo
    senão corrente  $\leftarrow$  próximo somente com probabilidade  $e^{\Delta E/T}$ 

```

Figura 2.6: Algoritmo de Têmpera Simulada (Russell e Norvig, 2009).

O principal procedimento do algoritmo consiste em um laço que gera aleatoriamente, em cada iteração, um único vizinho da solução atual. A seguir são mostradas as principais etapas do algoritmo:

1. inicia aleatoriamente um ponto x (ponto inicial) no espaço de busca;

2. a cada nova iteração, seleciona um ponto novo x' que esteja na vizinhança da solução inicial x , $x' \in N(x)$;
3. determina a energia de cada uma das configurações $E(x)$ e $E(x')$ e verifica a variação da energia do sistema: $\Delta E = E(x') - E(x)$;
4. se $\Delta E \geq 0$; então x' torna-se o ponto corrente ($x \leftarrow x'$); caso contrário a probabilidade de x' ser selecionado como ponto corrente é dada por um caso particular da distribuição de Boltzmann-Gibbs, ou seja, $P(\Delta E) = \exp(-\frac{\Delta E}{T})$;
5. o algoritmo termina quando algum critério de parada seja atingido, quando nenhuma melhora significativa seja alcançada ou quando a temperatura T atingir seu valor mínimo.

Essa técnica tem como principal característica o fato do algoritmo testar soluções diferentes da solução atual e oferecer mais independência do ponto inicial da pesquisa. Zhang e Wang (2011) e Bin et al. (2007) são exemplos de trabalhos que usam Têmpera Simulada para geração de dados de teste.

2.4.3.4 Algoritmo Genéticos

Como o tema está relacionado com este trabalho, a seguir é apresentado, com detalhes, os principais conceitos relacionados ao algoritmo evolutivo Algoritmo Genético. Provavelmente os Algoritmos Genéticos (AG) – proposto por Holland (1975) que tinha como objetivo inicial estudar formalmente o fenômeno da adaptação e desenvolver uma forma de importá-los para sistemas computacionais (Mitchell, 1998) – sejam a classe mais conhecida dos Algoritmos Evolucionários. Os AG's são modelos computacionais que imitam os mecanismos da evolução natural para resolver problemas de otimização.

Os Algoritmos Genéticos são caracterizados aplicando conceitos da evolução das espécies conhecido como operadores genéticos como: genes, cromossomos, seleção, cruzamento e mutação e é usado para resolver problemas que possuem um amplo espaço de busca.

Indivíduo

Quando se passa o problema para a codificação, a fim de se obter uma solução, o cromossomo é representado por uma lista de caracteres, geralmente codificada em binário (Figura 2.7). Cada cromossomo é formado por genes, sendo que cada gene possui uma informação sobre a resolução do problema em questão.

Figura 2.7: Um cromossomo codificado em binário.

População de Cromossomos

Um conjunto de cromossomos ou indivíduos é chamado de população. A população nada mais é do que um aglomerado de indivíduos que são possíveis candidatos à solução do problema e que, geralmente, inicialmente é gerada de forma aleatória. A cada geração de uma nova população o número de indivíduos matem-se o mesmo e sua evolução ocorre após a aplicação de operadores de cruzamento e mutação.

Função Fitness

Uma função de aptidão ou *fitness* é uma função de avaliação específica inerente a cada problema, que determina qual indivíduo é o mais apto daquela população. Esta função de avaliação é extremamente importante para os Algoritmos Genéticos, uma vez que ela mostra o quanto o cromossomo está próximo da melhor solução ou uma solução aceitável.

Seleção

Uma decisão importante durante a aplicação do AG é como escolher os indivíduos que estarão presentes na próxima geração, cuja responsabilidade fica a cargo do método de seleção. O processo de seleção nada mais é do que a seleção de dois indivíduos pais para geração de dois indivíduos filhos que estarão presentes na próxima geração (de Castro, 2006). Geralmente, os indivíduos com o melhor valor de *fitness* têm maior probabilidade de serem selecionados para reprodução. A seguir são descritos alguns dos principais métodos para seleção de indivíduos.

Método de Seleção por Roleta

Um dos métodos para seleção de indivíduos mais usado e mais simples é o da Roleta. Neste método os indivíduos são selecionados utilizando-se uma roleta, semelhante a uma roleta utilizada em jogos de azar. Os indivíduos são representados nesta roleta a uma proporção equivalente ao valor de *fitness* alcançada por este. Dessa forma os indivíduos com valores de *fitness* maiores recebem uma fatia maior na roleta, por outra mão os indivíduos com *fitness* menores recebem uma fatia menor na roleta. Para a seleção dos indivíduos a roleta é girada por um determinado número de vezes, definido pelo tamanho da população e então um indivíduo é selecionado para a população intermediária (Goldberg, 1989; Holland, 1975).

A Figura 2.8 ilustra uma roleta criada a partir dos valores de aptidão de uma população com quatro indivíduos. Na figura a roleta é dividida em 24 seções de 15 graus cada e a cada indivíduo é atribuído uma fatia da roleta formada pelo número de seções correspondente ao valor de seu *fitness*. Uma desvantagem desse método de seleção é que devido a primazia dos indivíduos com maiores valores de *fitness* pode ocorrer convergência prematura.

Método de Seleção por Torneio

Na Seleção por Torneio n indivíduos são escolhidos de forma aleatória para participarem de um torneio. Neste torneio os indivíduos competem entre si para que seja decidido quais farão parte da próxima geração. O indivíduo com o maior valor de *fitness* é então escolhido para população intermediária.

O processo de seleção dos indivíduos para o torneio pode ocorrer com ou sem reposição e é repetido até que a população intermediária seja preenchida. O valor de n geralmente é igual a três. A Figura 2.9 ilustra o método de seleção por Torneio com n igual a três.

Figura 2.8: Método de seleção por Roleta.

Figura 2.9: Método de seleção por Torneio com n igual a três.

Operadores Genéticos

Os operadores genéticos são a base do funcionamento dos Algoritmos Genéticos permitindo que uma população inicial se movimente por um dado número de iterações (gerações) e consiga gerar populações sucessivas com novos indivíduos possuindo genes mais adaptados da geração anterior. Assim, cada nova população gerada é totalmente nova, porém possuindo características de bons indivíduos das gerações anteriores.

Crossover

Para realizar o cruzamento é necessário um método de seleção que identificará os melhores indivíduos da população de acordo com seu valor de aptidão. Um dos métodos de cruzamento mais utilizado é conhecido como “cruzamento com um ponto de corte” que consiste em realizar o cruzamento por meio de um ponto de corte no cromossomo. A partir do ponto de corte selecionado ocorre a troca dos genes gerando uma nova descendência. A Figura 2.10 representa o cruzamento com um ponto de corte. A partir do ponto de corte selecionado ocorre a troca do material genético gerando dois indivíduos possuindo as características dos pais.

Figura 2.10: Representação de cruzamento com um ponto de corte.

Mutação

O operador de mutação tem como objetivo a alteração aleatória de um gene do cromossomo. Dessa forma, uma mudança na constituição cromossômica do indivíduo ocorre com o aparecimento de um caractere que não existia nas gerações passadas. Com isso é garantido uma diversificação genética entre os indivíduos e é alcançado um número maior de candidatos com possível solução ótima. A Figura 2.11 exemplifica uma operação de mutação.

Figura 2.11: Exemplificação de um operador de mutação.

Estratégias Aplicadas a AG's

Os Algoritmos Genéticos podem usar estratégias, durante o processo de evolução para modificar seu comportamento e resultado, que auxiliam na melhoria do processo. A seguir são apresentadas algumas dessas estratégias. A técnica Elitismo garante que durante o processo de evolução os melhores indivíduos do processo não se percam. Como os AG's são estocásticos, existe a probabilidade que bons indivíduos de uma determinada população não sejam selecionados para reprodução. Com o uso dessa técnica os indivíduos com melhor escore de *fitness* é copiado da população intermediária para a próxima população (De Jong, 1975). Com a estratégia de Eliminação de Duplicatas, durante o processo de evolução o AG, ignora os indivíduos duplicados, gerando um novo indivíduo para substituí-lo. Caso essa eliminação não seja feita, pode-se prejudicar o resultado final, pois a presença de indivíduos duplicados ocasiona a perda da diversidade na população (Davis e Mitchell, 1991).

A estratégia de Busca Tabu é uma meta-heurística para busca local que se utiliza de estruturas especializadas de memória para evitar problemas com ótimos locais e alcançar

um resultado balanceado entre intensidade e diversidade na busca. A Busca Tabu implementa uma lista, chamada Lista Tabu, que proporciona um mecanismo de memorização, evitando que o processo de busca entre em um ciclo repetitivo. Quando um indivíduo é avaliado ele é adicionado na Lista Tabu evitando assim, que a avaliação se repita a menos que satisfaça condições para ignorar esta restrição, chamadas de restrições Tabu (Ombuki et al., 2002).

Outra estratégia aplicada a AG's é a Similaridade. O objetivo dessa estratégia é reduzir o valor de aptidão de indivíduos similares dentro de uma população. Essa estratégia consiste em obter uma medida de similaridade entre os indivíduos da população e penalizar o valor de aptidão dos indivíduos conforme essa medida de similaridade. Quanto maior a medida de similaridade de um determinado indivíduo com relação aos outros indivíduos da população, maior a redução em seu valor de aptidão função de *fitness*. A partir do valor de similaridade do indivíduo, basta intervir no processo normal de avaliação dos indivíduos em um AG tradicional, punindo o valor de aptidão dos indivíduos com sua respectiva medida de similaridade. Com isso, tem-se um controle de crescimento das espécies dentro da população proporcionando a diversidade da população (Goldberg e Richardson, 1987).

Funcionamento

Um Algoritmo Genético começa com um conjunto de k indivíduos, gerados aleatoriamente, chamados **população**. Cada indivíduo, é representado como uma cadeia sobre um alfabeto finito. Esses estados são avaliados pela função de avaliação ou **função de *fitness***. Uma função de *fitness* deve retornar valores mais altos para estados melhores. Na Figura 2.12 é apresentado um algoritmo simples de Algoritmo Genético.

```

função ALGORITMO-GENÉTICO (população, FN-FITNESS) retorna um indivíduo
  entradas: população, um conjunto de indivíduos
             FN-FITNESS, uma função que mede a adaptação do indivíduo

  repita
    nova_população  $\leftarrow$  conjunto vazio
    para  $i \leftarrow 1$  até TAMANHO (população) faça
       $x \leftarrow$  SELEÇÃO-ALEATÓRIA (população, FN-FITNESS)
       $y \leftarrow$  SELEÇÃO-ALEATÓRIA (população, FN-FITNESS)
      filho  $\leftarrow$  REPRODUZ ( $x, y$ )
      se (pequena probabilidade aleatória) então filho  $\leftarrow$  MUTAÇÃO(filho)
      adicionar filho a nova_população
    população  $\leftarrow$  nova_população
  até algum indivíduo estar adaptado suficiente ou até ter decorrido tempo suficiente
  retornar o melhor indivíduo em população, de acordo com FN-FITNESS

```

```

função REPRODUZ( $x, y$ ) retorna um indivíduo
  entradas:  $x, y$ , indivíduos pais

   $c \leftarrow$  COMPRIMENTO ( $x$ )
   $c \leftarrow$  número aleatório de 1 a  $n$ 
  retornar CONCATENA(SUBCADEIA ( $x, 1, c$ ), (SUBCADEIA ( $y, c + 1, n$ ))

```

Figura 2.12: Pseudocódigo de um Algoritmo Genético (Russell e Norvig, 2009).

A formulação de um Algoritmo Genético simples pode ser resumida nos seguintes passos:

1. define-se uma população inicial, geralmente formada por indivíduos aleatórios e o tamanho é baseado no tipo e na complexidade do problema;
2. cálculo para a avaliação de aptidão de cada indivíduo (fitness);
3. processo de seleção é aplicado para selecionar os melhores indivíduos da população para reprodução e criação de novos candidatos à solução, denominado de nova geração;
4. aplica-se o operador de cruzamento nos candidatos selecionados;
5. aplica-se o operador de mutação nos descendentes;
6. o Algoritmo Genético é finalizado quando a solução satisfatória for encontrada, quando for alcançado um determinado número de gerações ou até que o algoritmo não consiga encontrar a solução desejada.

2.5 Considerações Finais

Neste capítulo foram apresentados conceitos básicos de testes de software. Também foram apresentados os principais conceitos de programação concorrente, os paradigmas de programação concorrente passagem de mensagem e memória compartilhada e suas formas particulares de realizar comunicação e sincronização. Quanto ao teste de programas concorrentes, foram apresentadas abordagens de modelos e tipos de critérios para o teste desses programas como a classificação de defeitos é importante quando se propõe um novo critério de teste. Neste capítulo também foram apresentadas as principais técnicas existentes para geração de dados de teste. Entre essas técnicas o Algoritmo Genético foi dado uma ênfase maior, pois tem sido aplicado em geração para programas sequencias com bons resultados e será utilizada no escopo deste trabalho. No próximo capítulo são apresentados trabalhos relacionados à geração de dados que usam algoritmos de busca para realizar tal tarefa.

Geração de Dados de Teste Usando Meta-heurística

3.1 Considerações Iniciais

Neste capítulo é apresentada uma visão geral dos trabalhos relacionados com a geração de dados de teste usando técnicas meta-heurísticas. Nem todos os trabalhos envolvem programação concorrente devido ser uma área ainda pouca explorada. No entanto, são descritos algumas contribuições para programas sequenciais, as quais serviram como base para a proposta deste trabalho.

Existem na literatura alguns trabalhos (Eytani, 2006; Hrubá et al., 2012; Krena et al., 2011; Letko, 2010) que usam meta-heurísticas para inserção de defeitos para teste em programas concorrentes. Como a proposta deste mestrado é sobre geração de dados, esses trabalhos não são detalhados nessa seção.

Este capítulo está organizado da seguinte forma: na Seção 3.2 são apresentados trabalhos relacionados à área de geração de dados. Na Seção 3.3 são apresentadas as considerações finais deste capítulo.

3.2 Trabalhos Sobre Geração de Dados de Teste

Existem na literatura muitos trabalhos que utilizam Algoritmos Genéticos para a geração de dados teste em programas sequenciais. A maioria deles visam à geração para critérios baseados em fluxo de controle (Jones et al., 1996; Michael et al., 2001; Pargas et al., 1999); alguns para critérios baseados em fluxo de dados (Bueno et al., 2001; Ferreira

e Vergilio, 2005); e outros focalizam em abordagens relacionados ao critério análise de mutantes (Bottaci, 2001; Ferreira e Vergilio, 2005). Esses trabalhos podem ser classificados, de acordo com a função de avaliação utilizada (Delamaro et al., 2007a), em duas categorias:

- **Função de avaliação baseada na cobertura:** a função de avaliação nos trabalhos pertencente a esta classe é realizada com base na cobertura de cada dado de teste em trabalhos. Os indivíduos na população codificam valores possíveis às variáveis de entrada do programa em teste. Ferreira e Vergilio (2005) é um exemplo de trabalho que usa esse tipo de função.
- **Função de avaliação orientada a um objetivo:** nesta classe de trabalhos, a função de avaliação é especificada de acordo com um objetivo, que geralmente se trata da cobertura de um elemento particular. As abordagens desta categoria geralmente consideram a execução dinâmica de programas e um objetivo que orienta a função de avaliação. Bueno et al. (2001) é um exemplo de trabalho que usa esse tipo de função.

O artigo de Korel et al. (1992) apresenta uma abordagem para geração de dados de teste que é baseada na execução atual do programa em teste, um “escalonador” em tempo de execução, métodos de minimização e análise dinâmica de fluxo de dados. Os dados de teste são gerados a partir de valores reais das variáveis de entrada do programa em teste. Quando o programa é executado o fluxo de controle é monitorado. Se durante a execução um fluxo “indesejado” é observado (por exemplo, o caminho atual não corresponde ao caminho previamente selecionado), então uma função de avaliação é associada ao ramo que está permitindo tal comportamento para gerar dados de teste que fazem com que o caminho correto seja percorrido. Essa função é usada pelo algoritmo de busca que aloca valores de entrada para o ramo executado. A análise de fluxo de dados dinâmica é usada para determinar quais as variáveis de entrada são responsáveis pelo comportamento “indesejado”. Essa informação ocasiona uma melhora de tempo no processo de geração de dados.

Inicialmente, uma sequência de eventos, que garante que o caminho vai ser executado, é derivado do caminho selecionado, então essa sequência é usada pelo “escalonador” para garantir que o caminho correto será percorrido. O processo de geração de dados apresentada nesse artigo é orientado a caminho, ou seja, primeiro um caminho do programa é selecionado, depois uma entrada que resulta na execução desse caminho é derivada. Se o dado de entrada não é encontrado o próximo caminho é selecionado. Esse processo continua até que um dado de entrada para o caminho selecionado seja encontrado ou o tempo limite é atingido.

Ferreira e Vergilio (2005) propõem um ambiente para geração de dados chamado TSD-Gen (*Test Data Set Generator*) que fornece mecanismos de hibridação e de memorização para melhorar o desempenho do Algoritmo Genético. Esses mecanismos são baseados em Lista Tabu e métrica de singularidade. O objetivo da abordagem é a geração de dados para cobertura dos critérios estruturais: *all-nodes*, *all-edges*, *all-potential-uses*, *all-potential-uses/du* e *all-du-paths*, apoiado pela ferramenta Poketool (Chaim, 1991); e

critério baseado em defeito apoiado pela ferramenta Proteum (Delamaro, 1993). As ferramentas são responsáveis pela geração dos elementos requeridos e pela avaliação da cobertura.

O ambiente proposto é composto por quatro módulos principais: *GenPopulation*, *Evaluator*, *Evolver* e *Initial*. Todas as informações geradas pelos módulos são salvas em arquivos.

O módulo *GenPopulation* é responsável pela geração da população e codificação de cada indivíduo de acordo com um critério. Os indivíduos são representados por uma concatenação de blocos e cada bloco está relacionado a uma entrada do programa. A população inicial pode ser tanto gerada aleatoriamente quanto fornecida pelo testador.

O módulo *Evaluator* é responsável pela avaliação de cada indivíduo usando o módulo de avaliação da ferramenta de teste correspondente. Esse módulo calcula a cobertura de cada indivíduo. Cada indivíduo é convertido em uma entrada para o programa sob teste no módulo *GenPopulation*. A lista de elementos cobertos é salva em um arquivo e posteriormente é usada pela função de avaliação. A função de avaliação utilizada (3.1) é dada pela cobertura do critério a ser satisfeito e pelo conjunto de elementos cobertos por indivíduo na população, sendo que cada indivíduo representa um dado de teste. Essa função é calculada pela lista obtida do módulo *Evaluator* que é da forma:

$$fitness = \frac{number - of - covered - elements * 100}{number - of - required - elements} \quad (3.1)$$

Baseada nessa função *fitness*, os indivíduos são selecionados e os operadores genéticos são aplicados.

O módulo *Evolver* é responsável pelo processo de evolução e pela aplicação dos operadores genéticos. O processo de evolução terminará quando o número de gerações, pré-determinado, for atingido ou quando a cobertura desejada é obtida.

O Módulo *Initial* é responsável por receber a configuração inicial do testador — por meio de um arquivo. O arquivo é composto basicamente por três seções: seção inclui parâmetros da ferramenta de teste correspondente; seção fornece informação para a execução do programa em teste; seção está relacionada com o processo de evolução e será usada pelo módulo *Evolver*. Nela contém taxa de mutação e cruzamento, tamanho da população, número máximo de gerações e quantidade de indivíduos que evoluirão por *fitness*, elitismo e similaridade.

Bueno et al. (2011) apresentam um novo método para geração de dados chamado DOTG (*Diversity Oriented Test Data Generarion*) que usa o conceito de diversidade de conjunto de teste. Os pesquisadores definem um domínio de entrada diversificado DOTG-ID, que considera a distância entre os dados de teste no domínio de entrada do programa para calcular um valor de diversidade para o conjunto de teste. A motivação para o desenvolvimento dessa técnica foi organizar o domínio de entrada de forma sistemática e consequentemente melhorar a qualidade do teste. As técnicas meta-heurística Têmpera Simulada, Algoritmo Genético e Repulsão Simulada (do inglês *Simulated Tepulsion*) são usadas para geração dos conjuntos de teste para o DOTG-ID.

A técnica Repulsão Simulada é inspirada em sistemas de partículas dinâmicas que são aplicadas a problemas de vários contextos como, por exemplo, problemas de otimização não-lineares que são resolvidos por comportamentos sociais. Em vez de usar a metáfora de comportamento social os autores usaram a metáfora de comportamento físico usando sistemas de partículas. Dessa forma, simularam um sistema físico que por conta de suas características, faz com que cada dado de teste (uma partícula no sistema) espontaneamente encontre seu lugar no domínio de entrada do programa para maximizar a diversidade do conjunto de casos de teste. Cada dado de teste no conjunto recebe uma carga elétrica positiva (Q), cujo valor inicial é um parâmetro da Repulsão Simulada. Essa carga faz com que cada dado de teste crie um campo elétrico em volta de si mesmo. Esse campo elétrico resulta em forças aplicadas no dado teste que causam sua movimentação dentro do domínio de entrada.

O algoritmo trabalha com um conjunto de teste NTS (*Number of Test Set*) = $T (T_1, T_2, \dots, T_{NTS})$. Cada conjunto de teste possui um conjunto de dados NTD (*Number of Test Dada*) = $T (T_1, T_2, \dots, T_{NTD})$ e cada conjunto de dados possui NIV (*Number of Input Variable*) = $T (T_1, T_2, \dots, T_{NIV})$. NTS , NTD , NIV . A descrição de cada variável de entrada x são parâmetros para o algoritmo Repulsão Simulada.

Usando um gerador de números aleatórios uniforme, o algoritmo gera um conjunto de teste NTS aleatório. Cada dado de teste presente no conjunto de teste representa uma partícula no domínio de entrada D . O valor de cada variável de entrada do dado de teste representa a posição do dado de teste em relação a D . Cada conjunto de teste T é uma partícula de sistema que evolui de forma independente e simultaneamente de outros conjuntos de teste. A cada iteração do algoritmo Repulsão Simulada, todos os dados de teste de todos os conjuntos têm sua posição atualizada.

À medida que a busca progride, as forças resultantes aplicadas a cada dado de teste diminui assim como quantidade total de movimentos. No estágio final, o sistema de partícula (conjunto de teste) alcança um estado próximo ao equilíbrio das forças e os movimentos dos dados de teste tendem a parar, alcançando a estabilização do valor de diversidade do conjunto de teste.

Louzada et al. (2012) propõem um Algoritmo Genético elitista para geração e seleção de dados de teste para teste de mutação para diferentes *benchmarks*. O escore de mutação obtido por meio da execução de programas mutantes gerados a partir de *benchmarks* é usado como função de avaliação. A ferramenta MuJava foi usada para gerar programas mutantes que interagem com o AG.

O método de seleção usado pelo AG foi o seleção de torneio que chama dois indivíduos selecionados aleatoriamente da população para participar do cruzamento. O cruzamento é feito nos indivíduos que possuem melhor avaliação. O Algoritmo Genético proposto usa uma estratégia elitista em que somente os melhores indivíduos são escolhidos para a próxima geração.

A função de avaliação usada no trabalho (3.2) é a soma do número da taxa de mutantes mortos de cada tipo, multiplicado pelos seus pesos. Os pesos w_1 , w_2 e w_3 usado na *fitness*, são respectivamente os pesos atribuído a cada tipo de operador de mutação, que nesse caso são: operador de mutação aritmético, operador de mutação de lógica e operador de mutação relacional. A quantidade de mutantes mortos por tipo são armazenados nas

variáveis $MutA$, $MutL$ e $MutR$ que equivalem, respectivamente, aos operadores de mutação aritmético, lógico e relacional.

$$FF = w1 * \frac{MutA}{MutKill} + w2 * \frac{MutL}{MutKill} + w3 * \frac{MutR}{MutKill} \quad (3.2)$$

Para calcular a função de avaliação inicialmente é identificada a proporção de mutantes mortos de acordo com cada tipo de operador de mutação (aritmético, lógico e relacional), em seguida é multiplicado por seus respectivos pesos ($w1$, $w2$ e $w3$) e finalmente a soma da taxa de mutação é realizada para encontrar o valor da função de avaliação.

Ghiduk et al. (2007) propõem uma técnica para geração de dados que usa Algoritmo Genético. A técnica aplica o conceito de relação de dominância entre os nós para definir uma função de avaliação multi-objetivo para avaliar a geração de dados. Uma árvore de dominância $DT(G) = (V, E)$ é um grafo direcionado em que um nó distinto n_0 , chamado raiz, é a cabeça das arestas. Cada nó n , exceto o nó n_0 , é a cabeça de apenas uma aresta e existe um (único) caminho (caminho de dominância $dom(n)$) da raiz n_0 para cada nó n . O algoritmo usa a versão instrumentada do programa em teste (P), o conjunto de teste requerido ($TestReq$) pelo critério de fluxo de dados (todos-usos) e a árvore de dominância (DT) como entrada. A população inicial ($InitPop$) é obtida aleatoriamente ou é fornecida pelo testador. Os parâmetros do Algoritmo Genético como, taxa de cruzamento, taxa de mutação e tamanho da população são obtidos por meio do testador. Assim, o algoritmo retorna o conjunto de dados de teste que cobre $TestReq$.

Andreou et al. (2007) propõem um *framework* que utiliza um grafo de fluxo de dados com um gerador de casos de teste que usa Algoritmo Genético para gerar automaticamente dados de teste baseados em critérios de fluxo de dados. O *framework* é composto por um analisador de programa e um gerador de dados de teste. O analisador de programa foi construído em dois subsistemas: analisador que funciona em tempo de execução e analisador que funciona estaticamente. O analisador que funciona estaticamente não precisa que o programa em teste seja executado, enquanto que o analisador em tempo de execução analisa e avalia o programa durante sua execução. O principal objetivo desses analisadores é extrair do programa informações essenciais para a criação do grafo de fluxo de dados. O gerador de dados de teste é baseado em um Algoritmo Genético que é usado para pesquisar o espaço de entradas e determinar o melhor conjunto de dados de teste. Cada gene do AG codifica uma estrutura de dados que encapsula informações de parâmetros de entrada distintos do programa. Essa estrutura contém os valores iniciais e atuais dos parâmetros e seus limites, ou seja, seus valores máximos e mínimos. O cromossomo, que é um conjunto de genes, codifica todo o caso de teste. A função de avaliação é guiada pelas características do grafo de fluxo de controle do programa em teste. Essa função leva em consideração não somente o número de caminhos cobertos, mas também quão efetivo é o cromossomo comparado com o resto dos cromossomos da atual população.

Zhang e Wang (2011) propõem uma abordagem para geração de dados de teste que faz a combinação de um Algoritmo Genético adaptativo com Têmpera Simulada. A função de avaliação, proposta por Lin e Yeh (2001), é conhecida como similaridade que quantifica precisamente a distância entre dois caminhos dados. Dado um caminho pretendido e o

caminho atual, a similaridade é calculada por meio das arestas em cascata de cada caminho que está sendo comparado. Depois de aplicar a taxa de cruzamento e de mutação, os cromossomos já não são mais os mesmos, assim é difícil garantir que esses são melhores que os originais. Dessa forma, os autores usam Têmpera Simulada para ajudar nessa decisão. O algoritmo de Têmpera Simulada é inserido no Algoritmo Genético para decidir a permanência ou não de novos indivíduos na população.

Tian e Gong (2014) apresentam uma abordagem de geração de dados para cobertura de caminhos de programas paralelos com passagem de mensagem com uso de Algoritmo Genético. Para isso, o problema de geração é primeiramente modelado com um problema de otimização de um único objetivo e de acordo a correlação entre os caminhos dos processos e suas entradas associadas um novo Algoritmo Genético é proposto para gerar os dados de teste. No início do processo de geração dos dados, são identificados todos os caminhos presentes no programa e os caminhos equivalentes associados a ele. Neste mesmo momento os caminhos que não foram cobertos também são identificados. A cada nova interação os dados de teste (indivíduo) são executados em todos os caminhos. Se um novo caminho percorrido pertence ao conjunto de caminhos não cobertos, o dado de teste é visto com um bom dado e o caminho percorrido e seus caminhos equivalentes associados são retirados do grupo de caminhos não cobertos. Se o conjunto de caminhos não cobertos estiver vazio o algoritmo é finalizado, caso contrário a função de *fitness* é ajustada de acordo com o caminho percorrido. Os autores não relacionam como tratam o problema de não determinismo.

3.3 Considerações Finais

Neste capítulo foram apresentados alguns trabalhos de geração de dados, com uso de várias meta-heurísticas, que estão relacionados com o tema deste mestrado.

No próximo capítulo é apresentada a abordagem de geração de dados de teste para programas concorrentes proposta neste projeto, seguida da descrição de como essa abordagem foi adicionada às demais funcionalidades da ferramenta ValiMPI.

Abordagem de Geração de Dados Proposta

4.1 Considerações Iniciais

A geração automática de dados de teste é de grande importância para o processo de desenvolvimento de software, pois com a automatização dessa etapa de teste é reduzido tempo e esforço. Existem várias ferramentas disponíveis para geração de dados que usam técnicas de meta-heurísticas, todavia são aplicadas a programas sequenciais.

Sabe-se que o teste para programas concorrentes é conduzido de forma diferente daquele para programas sequencias. Essas diferenças podem ser observadas em características como o não determinismo nesses programas. Essas características precisam ser levadas em consideração durante a atividade de teste.

Este capítulo apresenta uma abordagem, chamada ValiGen, para geração automática de dados de teste usando Algoritmo Genético para programas concorrentes em MPI. A abordagem apresentada é baseada no trabalho de Ferreira e Vergilio (2005), no qual os autores apresentam uma ferramenta de apoio à geração automática de dados de teste voltada para programas sequencias. A abordagem apresentada neste trabalho foi implementada como um módulo da ferramenta de teste ValiMPI estendendo assim suas funcionalidades.

O capítulo está organizado da seguinte maneira: Na Seção 4.2 é apresentada a abordagem para geração de dados, seus mecanismos e como o não determinismo é tratado durante a geração. Na Seção 4.3 é apresentada a arquitetura do módulo proposto e como o mesmo está integrado à ferramenta ValiMPI e um exemplo de uso. Na Seção 4.4 são apresentadas as considerações finais deste capítulo.

4.2 Apresentação da Abordagem Proposta

A abordagem proposta neste trabalho combina conceitos de teste baseados em busca para geração dos dados e conceitos de teste de alcançabilidade para resolver o problema do não determinismo que é inerente ao teste de software em programas concorrentes.

Como visto no Capítulo 3 o uso de meta-heurísticas tem se mostrado uma ótima opção para geração de dados, pois estes algoritmos apresentam bons resultados para resolver problemas inerentes a esta atividade. Neste trabalho, a meta-heurística usada para geração dos dados foi Algoritmo Genético.

Algoritmo de Geração de dados

O Algoritmo Genético implementado usa estratégias de evolução e mecanismos de memorização para melhorar a geração de dados de teste. Este AG usa informações de cobertura de elementos requeridos para melhorar dados de teste para que sejam capazes de exercer novas sequências de sincronização.

População Inicial

No início do processo de geração, uma população inicial, contendo um conjunto de dados de teste, é gerada de forma aleatória ou por meio de um arquivo.

Para o Algoritmo Genético, cada dado de teste é um indivíduo na população. Assim, cada indivíduo é uma entrada do programa em questão. No indivíduo, cada argumento da entrada é representado como um bloco, que é uma codificação usada pelo AG, cujo formato depende do tipo do argumento.

Para codificação de um tipo de dado inteiro, utiliza-se uma máscara de seis posições. Uma posição para o sinal e cinco para o número, resultando em +00000. Para a codificação de tipos de dados *float*, utiliza-se uma máscara de nove posições. Uma posição para o sinal, cinco para o número, uma posição para o sinal da mantissa e duas posições para o valor da mantissa, resultando em +00000+00. A máscara para o tipo *double* é semelhante à máscara de *float*, porém possui uma capacidade maior refletindo em mais uma posição para a mantissa, resultando em +00000+000. A Tabela 4.1 apresenta exemplos de tipos de dados e suas codificações. Cada valor de entrada codificado é chamado de bloco.

Tabela 4.1: Codificação dos tipos de dados

Tipo	Máscara	Valor	Codificado
Inteiro	+00000	-543	-00543
Float	+00000+00	-543000	-00543+03
Double	+00000+000	-543000	-00543+003

A concatenação desses blocos representam um indivíduo na população. Dessa forma, se o programa possui entradas do tipo inteiro e do tipo real, o indivíduo será representado pela concatenação desses blocos. Essa representação (Figura 4.1) é importante pois durante a etapa de evolução do AG os tipos de dados são levados em consideração para não haver erros durante a geração de novos dados.

Figura 4.1: Os indivíduos são representados por uma concatenação de blocos e cada bloco está relacionado a uma entrada do programa.

Avaliação da População

A avaliação da população é feita de duas maneiras: por valor de *fitness* e por bônus de similaridade.

Na avaliação por valor de *fitness*, cada indivíduo da população é executado na ValiMPI levando em consideração um critério de teste estabelecido. A execução dos dados é feita de forma controlada pela ValiMPI, tratando o não determinismo (para melhor entendimento, a parte que diz respeito ao não determinismo durante este processo é apresentada mais adiante). Dessa forma, a ferramenta de apoio gera os elementos requeridos cobertos por aquela execução. Com a execução de todos os dados de teste daquela população, é possível saber quais elementos requeridos foram cobertos e inclusive por qual dado de teste. Assim uma função de avaliação ou *fitness* é calculada para cada indivíduo usando informações de quantidade de elementos requeridos cobertos.

O valor de *fitness* de um indivíduo x é obtido pela divisão dos requisitos cobertos por ele pela quantidade de requisitos de teste que precisam ser cobertos. Esse cálculo é feito pela Fórmula 4.1.

$$fitness(x) = \frac{numero_de_elementos_cobertos_x * 100}{numero_de_elementos} \quad (4.1)$$

A outra estratégia usada para avaliação dos dados é o bônus de similaridade. Esse bônus é um valor que cada indivíduo da população recebe por satisfazer requisitos de teste ainda não cobertos. Isso significa que se ele é capaz de cobrir um requisito novo, ele recebe uma bonificação, indicando sua efetividade. Esse bônus é calculado para cada elemento requerido x , a partir da informação de cobertura de todos os indivíduos sobre x (veja Fórmula 4.2). Se poucos indivíduos cobrem o elemento x , esse elemento recebe uma bonificação maior. O valor final de bônus de similaridade para cada indivíduo é a somatória de todos os bônus por ele conseguido. Dessa forma, durante a evolução, indivíduos com baixo valor de *fitness*, porém que cubram elementos requeridos inéditos, não são penalizados.

$$BonusSimilaridadeElemento(x) = 100 * \left(1 - \frac{numero_de_indivíduos_cobrem_x * 100}{numero_de_indivíduos} \right) \quad (4.2)$$

Evolução da População

Após os dados serem avaliados, o Algoritmo Genético aplica estratégias de evolução dos dados avaliados para então formar a próxima população de dados de teste. As estratégias utilizadas são *Fitness*, Elitismo e Similaridade.

A evolução por *Fitness* é um método tradicional de evolução no qual a seleção dos indivíduos pais é feita com base no *score* de *fitness* por estes alcançados. Um método de seleção é usado para fazer a eleição dos indivíduos pais.

O método de seleção escolhido foi Seleção por Roleta, no qual os indivíduos são representados nessa roleta por uma porção equivalente ao seu valor de *fitness*. Ou seja, quanto maior o *fitness* de um indivíduo, maior é a porção que este ocupa na roleta e maior é chance de ser selecionado. Um número aleatório é sorteado entre zero e o somatório do *fitness* de todos os indivíduos. Este número representa o ponto selecionado da roleta, que corresponde a um indivíduo.

Depois da seleção, os operadores genéticos *crossover* e mutação são aplicados.

O *crossover* entre dois indivíduos pais ocorre entre blocos e não sobre a representação do indivíduo como um todo. Dessa forma, quando o cruzamento de indivíduos compostos de tipos de dados diferentes acontece, não ocorre problema de cruzamento entre tipos de dados diferentes.

A mutação ocorre logo após a etapa de *crossover* dos dois indivíduos, independente de ter ocorrido o cruzamento ou não. Da mesma forma que ocorre no operador *crossover*, o operador mutação é aplicado sobre blocos na representação dos indivíduos.

Se os indivíduos resultantes da etapa da aplicação dos operadores genéticos já existirem na população, estes são descartados e o processo de geração continua até que a quantidade de indivíduos configurados para geração seja alcançada.

A estratégia de evolução com elitismo é uma estratégia que introduz um indivíduo na próxima geração baseado na lista ordenada pelo valor de *fitness*. Com isso é garantido que bons indivíduos com altos valores de *fitness* estarão na próxima população. Se o indivíduo selecionado já estiver na população, o próximo é então selecionado e assim por diante.

A estratégia por Similaridade é baseada no bônus de similaridade alcançada por cada indivíduo da população. Como cada indivíduo possui um bônus de similaridade, uma lista ordenada com esses valores é obtida. Similarmente a estratégia que usa elitismo, os indivíduos que obtiverem os maiores valores de similaridade são selecionados para a próxima geração.

Além das estratégias de evolução apresentadas, um mecanismo para memorização durante o processo de evolução pode ser utilizado. Este mecanismo, chamado de lista Tabu, é executado ao fim de todas as estratégias de evolução citadas. Um indivíduo x é adicionado na lista se este cobrir um elemento ainda não coberto pelo restante dos indivíduos presentes na lista. O tamanho da lista é dado pelo número de elementos requeridos do critério de teste. Esse mecanismo funciona como uma memória com os melhores indivíduos obtidos durante todo o processo de evolução.

Não Determinismo

Como já citado, o não determinismo é uma característica presente nos programas concorrentes que precisa ser levada em consideração no teste desses programas. Assim, ao lidar com geração de dados para programas concorrentes, é necessário que os mesmos percorram todas as sequências de sincronização que podem ser executadas por esse dado de teste.

Na abordagem proposta, o não determinismo também é levado em consideração. Para tanto, o módulo Vali_Sync da ferramenta ValiMPI apresentado no Capítulo 2 é utilizado para execução controlada do programa sob teste. Funciona da seguinte forma: ao ser gerado um novo dado de teste, este é executado no módulo Vali_Exec, da ferramenta ValiMPI, que gera os *traces* percorridos por este dado no programa. Em seguida o módulo Vali_Sync é executado, o qual analisa esses *traces* percorridos, identificando novas variantes (caminhos com novas sequências de sincronização) desses *traces*. De posse dessas novas variantes, o dado é reexecutado percorrendo esses novos caminhos. Dessa forma, é possível manter, para cada dado de teste, as suas possíveis saídas e sequências de execução. Essa informação é importante para a evolução da geração dos dados de teste. A Figura 4.2 ilustra esse cenário.

Quando um dado de teste é executado na ferramenta, todas as possíveis sequências de sincronizações alcançadas por este dado é forçada a acontecer pelo módulo de execução controlada presente na ValiMPI. Dessa forma, o dado gerado possui uma probabilidade maior de encontrar defeitos presentes na comunicação entre processos.

Figura 4.2: Não determinismo na geração de dados.

Funcionamento da Abordagem Proposta

As etapas de geração dos dados descritas anteriormente são sumarizadas no algoritmo

1. Primeiro, os dados iniciais (T_i) são gerados de forma aleatória ou é fornecida por

meio de um arquivo. Esses dados são executados na ValiMPI que por sua vez executa os módulos Vali_Exec e Vali_Sync e Vali_Eval. Com a execução dos dados na Vali_Sync todas as possíveis sincronizações que o dado pode percorrer são executadas. A última etapa da ValiMPI é a execução do módulo Vali_Eval que retorna ao algoritmo de geração todos os elementos requeridos cobertos pelo dado de teste. Depois, baseado no índice de cobertura do indivíduo, o AG calcula o valor de *fitness* e similaridade de todos os indivíduos da população. Os indivíduos da população são somente adicionados à lista Tabu se obedecerem às suas restrições. Em seguida, a população é evoluída (T_{i+1}) empregando as estratégias de *Fitness*, Similaridade e Elitismo. Este processo continua até ser atingido um critério de parada. Ao final do processo de evolução, é retornada a lista Tabu com os melhores indivíduos de todo o processo e a geração contendo a melhor população.

Algoritmo 1 Etapas durante a geração dos dados

```

1: gera conjunto de dados de teste inicial ( $T_i$ )
2: enquanto criterio de parada não atingido faça
3:   executa Vali_Exec ( $T_i$ ) ▷ executa o dado de teste e gera traces com sincronizações
4:   executa Vali_Sync ( $T_i$ )           ▷ gera demais sequencias de sincronizacao para  $T_i$ 
5:   executa Vali_Eval ( $T_i$ )           ▷ avalia os dados segundo um critério
6:    $f :=$  calcula fitness ( $T_i$ )
7:    $s :=$  calcula similaridade ( $T_i$ )
8:   armazena lista Tabu
9:    $T_{i+1} :=$  evolui dados de teste ( $(T_i), f, s$ )
10: fim enquanto
11: devolve lista Tabu
12: devolve melhor geração ( $T$ )

```

Para implementação da abordagem proposta, um módulo de geração de dados, chamado ValiGen, foi implementado. Este módulo foi integrado as demais funcionalidades da ferramenta da ValiMPI. A arquitetura, funcionamento e aspectos de implementação são descritos a seguir.

4.3 Arquitetura e Funcionamento da ValiGen

O módulo ValiGen foi desenvolvido na linguagem de programação Java e funciona em conjunto com a ferramenta ValiMPI descrita no Capítulo 2. A geração dos dados ocorre para os critérios Todos-Nos-R, Todos-Nos-S, Todas-Arestas-S, Todos-S-Usos, suportados pela ValiMPI.

O módulo ValiGen possui quatro submódulos principais: Central, Geração da População, Avaliação e Evolução. No módulo Central, um arquivo de configuração é utilizado para obtenção de informações que serão usadas pela ferramenta de apoio e pelo módulo de geração. Em seguida, o módulo Geração da População gera uma população inicial, composta de dados de teste, que pode ser de forma aleatória ou fornecida pelo testador. No módulo de Avaliação, essa população é executada na ValiMPI, analisando a cobertura dos elementos requeridos. A próxima etapa é realizada pelo módulo de Evolução, que

usa informações de cobertura obtidas pelos dados para aplicar estratégias para geração de novos dados. Esses novos dados são iterativamente executados na ferramenta ValiMPI, passando pelos módulos Avaliação e Evolução, até que um determinado critério de parada seja alcançado.

A arquitetura do módulo ValiGen é apresentada na Figura 4.3 onde é possível observar suas atividades principais e sua interação com os demais módulos da ferramenta ValiMPI.

Figura 4.3: Arquitetura do módulo ValiGen.

Nas subseções seguintes são apresentados com detalhes os principais componentes do módulo ValiGen.

Módulo Central

O módulo Central é responsável pela comunicação com os outros módulos sobre as configurações de execução pré-estabelecidas. As informações de configuração são lidas pelo módulo Central por meio de um arquivo de configuração. A interpretação do arquivo de configuração é a primeira etapa de execução deste módulo. Neste arquivo, estão contidas informações relacionadas sobre o programa em teste, como quantidade de entradas e tipos de dados das entradas; informações sobre o processo de geração, como taxa de *crossover* e taxa de mutação; e informações que serão utilizadas pelos demais módulos da ValiMPI,

como nome das funções presentes no programa e quantidade de processos paralelos a serem criados.

Para interpretação do arquivo de configuração os parâmetros passados ao arquivo devem obedecer as seguintes regras:

- cada parâmetro deve estar presente em uma única linha;
- cada linha de declaração de parâmetro deve estar no formato “#parâmetro = valor”.

Qualquer linha do arquivo de configuração que não obedece a essas regras é ignorada. Um exemplo de arquivo de configuração é apresentado na Figura 4.4.

```
#Criterio = Todos os Usos de Sincronismo
#NumeroArgumentos = 3;
#FormatoEntrada = III;
#ArquivoFonte = gcd.c;
#NomeFuncao = gcd;
#CoberturaCriterio = -1;
#VariacaoInt = 1/1000;
#TaxaCrossover = 0.9;
#TaxaMutacao = 0.5;
#TamanhoPopulacao = 50;
#NumeroGeracoes = 100;
#Elitismo = 10;
#Ineditismo = 10;
#Fitness = 30;
#AtivaTabu = 1;
#GeracoesComRepositorio = 90;
#NumeroProcessos = 4;
#Funcoes = "Master(0)" "Slave(1,2,3)";
```

Figura 4.4: Arquivo de configuração usado no exemplo.

As informações presentes no arquivo podem ser divididas em quatro categorias:

- informações que serão usadas pela ferramenta ValiMPI, como nome do arquivo fonte do programa, nome da função, critério de teste, números de processos concorrentes que serão utilizados no programa;
- informações que serão usadas pelo algoritmo de geração presente no módulo ValiGen, como taxa de *crossover*, taxa de mutação, quantidade de indivíduos que evoluirão por *Fitness*, Similaridade e Elitismo e número máximo de gerações.
- informações relacionadas ao programa em teste como, tipo do formato de entrada do programa, número de argumentos e nome do arquivo da população inicial, se necessário;
- informações relacionadas ao módulo de geração ValiGen como, habilitação do recurso de lista Tabu, número de geração que usam o recurso de repositório.

Como dito anteriormente, cada parâmetro deve ser seguido por um valor. A Tabela 4.2 lista cada parâmetro aceitável pelo arquivo de configuração e uma breve descrição dos valores que podem assumir.

Tabela 4.2: Descrição do parâmetros do arquivo de configuração e seus valores

Parâmetro	Obrigatório	Detalhes
#ArquivoFonte	Sim	Nome do programa a ser testado.
#NomeFuncao	Sim	Nome da função a ser testada
#Critério	Sim	Crítério de teste a ser utilizado
#CoberturaCritério	Sim	Valor α para a Cobertura desejável $0 \leq \alpha \leq 1$.
#TamanhoPopulacao	Sim	Tamanho γ da população, no qual $\gamma > 0$.
#NumeroGeracoes	Sim	Número máximo δ de gerações, $\delta \geq 0$
#TaxaMutacao	Sim	Taxa ϵ de mutação para o AG, $0 \leq \epsilon \leq 1$.
#TaxaCrossover	Sim	Taxa ε de crossover para o AG, $0 \leq \varepsilon \leq 1$.
#Elitismo	Sim	Quantidade ζ de indivíduos que evoluem por elitismo, $0 \leq \zeta \leq \gamma$ e $\zeta + \eta + \theta = \gamma$.
#Similaridade	Sim	Quantidade η de indivíduos que evoluem por similaridade, $0 \leq \eta \leq \gamma$ e $\eta + \zeta + \theta = \gamma$.
#Fitness	Sim	Quantidade θ de indivíduos que evoluem por fitness, $0 \leq \theta \leq \gamma$ e $\theta + \eta + \zeta = \gamma$.
#FormatoEntrada	Sim	Tipos de dados da entrada do programa. I-int, F-float, D-double.
#NumeroArgumentos	Sim	Número β de parâmetros de entrada esperados como argumento, no qual $\beta \geq 0$.
#ArquivoPopulacao	Não	Arquivo com dados semente para população inicial.
#AtivaTabu	Não	Ativar estratégia de lista Tabu.
#VariacaoInt	Não	Varição do tipo inteiro na forma $Lim_{inferior} / Lim_{superior}$. Default 0/10000
#VariacaoDouble	Não	Varição do tipo inteiro na forma $Lim_{inferior} / Lim_{superior}$. Default 0/10000
#VariacaoFloat	Não	Varição do tipo inteiro na forma $Lim_{inferior} / Lim_{superior}$. Default 0/10000
#GeracaoComRepositorio	Não	Quantidade de gerações que usará o repositório de cobertura.

O parâmetro #ArquivoFonte é usado para informar o nome do arquivo a ser testado seu uso é obrigatório, pois consiste de um parâmetro necessário.

O critério de teste é informado pelo parâmetro #Critério e deve sempre estar presente. Os valores possíveis para este parâmetro são os nomes dos critérios: Todos-Nos-R, Todos-Nos-S, Todas-Arestas-S, Todos-S-Usos.

O parâmetro #CoberturaCritério informa ao módulo uma cobertura aceitável para o critério. Assim a execução pode ser encerrada ao alcançar tal cobertura. Se configurado com o valor -1, o módulo executará até que o máximo de gerações seja alcançado.

O tamanho da população é informado com o parâmetro #TamanhoPopulacao e deve ser um valor maior que zero.

Para informar o número máximo de gerações é utilizado o parâmetro #NumeroGeracoes que deve ser um valor maior que zero. Este parâmetro pode ser configurado para zero, caso em que o processo de evolução não será realizado e apenas a população inicial será avaliada, ou seja, será usada geração aleatória.

Para indicar a taxa de mutação o parâmetro `#TaxaMutacao` é utilizado e deve conter um valor maior que 0 e menor que 1.

Para indicar a taxa de *crossover* o parâmetro `#TaxaCrossover` é utilizado e deve conter um valor maior que 0 e menor que 1.

Os parâmetros `#Elitismo`, `#Similaridade` e `#Fitness` informam a quantidade de indivíduos que estarão na próxima geração por valor de elitismo, similaridade e *fitness*, respectivamente. A soma dos valores desses três parâmetros deve ser do tamanho da população. Na prática, um valor atribuído a um desses parâmetros significa a quantidade de indivíduos presentes na próxima população que usa essa estratégia.

O tipo de dados que a entrada do programa em teste espera deve ser informado através do parâmetro `#FormatoEntrada`. Seu valor deve ser na ordem em que o programa os espera, sendo utilizada a convenção de I para *int*, F para *float*, D para *double*. Se o programa espera, por exemplo, um *int*, um *double* e um *int*, o conteúdo deste parâmetro será IDI.

O uso do parâmetro `#NumeroArgumentos` informa ao módulo a quantidade de entradas esperada pelo programa testado.

O parâmetro `#ArquivoPopulação` é opcional e possibilita o fornecimento de uma população inicial para o AG. O parâmetro `#AtivaTabu` ativa o recurso de memorização disponível no módulo. Este parâmetro é ativado com os valores "1" ou "sim" e desativado com qualquer outro.

Os parâmetros `#VariacaoInt`, `#VariacaoDouble`, `#VariacaoFloat` são úteis para limitar o intervalo de valores dos dados durante a geração. Este recurso pode ser interessante para determinados tipos de programas.

O parâmetro `#GeracoesComRepositorio` é utilizado para balancear o ganho de desempenho com a utilização do repositório de execuções. Este recurso evita que um indivíduo já avaliado seja avaliado novamente caso já esteja presente em alguma geração anterior. Seu valor é a quantidade máxima de gerações armazenadas nesse repositório dentro do processo de geração.

Módulo Geração da População

O módulo de geração da população é responsável por gerar a população inicial. Este módulo também é responsável pela codificação dessa população que será manipulada pelo AG para evolução da população.

Essa população inicial são dados de teste que serão codificados em forma de indivíduos para uso do AG. O módulo suporta duas formas para se obter a população inicial: aleatória ou via arquivo de entrada.

Na forma aleatória os indivíduos iniciais são gerados aleatoriamente a partir de informações extraídas do arquivo de configuração, como tipos de dados, e quantidade de argumentos presentes no programa. Já via do arquivo de entrada, os dados iniciais são obtidos por meio de um arquivo de texto, onde se encontram dados de teste relacionados ao programa. Cada linha do arquivo deve conter um dado de teste.

Em seguida, a população inicial é avaliada pelo módulo de Avaliação.

Módulo de Avaliação

No Módulo de Avaliação as estratégias de avaliação apresentadas na Seção 4.3 são utilizadas. Para tanto, dois arquivos são gerados.

A Figura 4.5 apresenta o arquivo usado para o cálculo do valor de *fitness*. Este arquivo contém informações da cobertura de elementos requeridos alcançada pelos indivíduos da população. Cada linha do arquivo corresponde a um indivíduo da população e a coluna corresponde aos elementos requeridos em relação ao critério de teste em questão. Na linha, a marcação “X” significa que o elemento foi coberto e a marcação “-” significa que não foi coberto.

```

1: X X - X X X - X X - - - X
2: X X - X X X - X X - - - X
3: X X X - X X - X X - - - X -
4: X X - X X X - X X - - - X
5: X X - X X X - X X - - - X
6: X X - X X X - X X - - - X
7: X X - X X X - X X - - - X
8: X X - X X X - X X - - - X
9: X X - X X X - X X - - - X
10: X X - X X X - X X - - - X

```

Figura 4.5: Arquivo de cobertura dos indivíduos de uma população.

Para calcular o bônus de similaridade outro arquivo é gerado com informações referentes a indivíduos que satisfazem um determinado requisito. Este arquivo é apresentado na Figura 4.6, o qual é uma transposta da matriz apresentada na Figura 4.5. Ou seja, as linhas representam os elementos requeridos e as colunas os indivíduos da população. A marcação “X” significa que o indivíduo em questão cobriu o requisito e a marcação “-” significa que não foi coberto.

Módulo de Evolução

No módulo de Evolução acontece a seleção e evolução dos indivíduos que estarão presentes na próxima geração da população. As estratégias de evolução por *Fitness* Similaridade e Elitismo são aplicadas. Cada uma dessas estratégias seleciona uma quantidade de indivíduos para a nova população, e essas quantidades são definidas no arquivo de configuração.

4.3.1 Arquivo de Saída

Ao final do processo de execução, o módulo gera um arquivo chamado “Populacao.res” contendo os dados de teste da geração que alcançou a melhor cobertura em todo processo. Se o módulo de geração estiver configurado para trabalhar com o mecanismo de memorização lista Tabu, um arquivo chamada “Tabu.res”, também é gerado contendo somente

```
1: X X X X X X X X X X
2: X X X X X X X X X X
3: - - - X - - - - -
4: X X X - X X X X X X
5: X X X X X X X X X X
6: X X X X X X X X X X
7: - - - - - - - - -
8: X X X X X X X X X X
9: X X X X X X X X X X
10: - - - - - - - - -
11: - - - - - - - - -
12: - - - - - - - - -
13: - - - X - - - - -
14: X X X - X X X X X X
```

Figura 4.6: Arquivo de cobertura dos elementos requeridos.

os dados de teste presentes na lista Tabu. Os dados presentes nesses dois arquivos estão em formato compatível com as entradas da ValiMPI.

Como dito anteriormente, a lista Tabu guarda somente os melhores indivíduos de todo o processo de evolução. Por conta disso, pode conter indivíduos de várias gerações, e não somente da melhor. Pode ser interessante ao testador ter conhecimento de qual geração veio cada dado, para reexecução. Dessa forma, um outro arquivo chamado “tabu.pop” também é gerado contendo informações de qual geração, e qual a posição nessa geração, de todos os dados presentes na lista Tabu.

Vale ressaltar que durante o processo de evolução, todos os dados de teste executados têm seus *traces* de execução armazenados. Isso permite a reexecução de um determinado dado de teste já executado de forma determinística.

A cobertura alcançada por cada geração durante todo o processo é armazenada no arquivo “resultado.tst”. No final deste arquivo, também estão presentes informações como cobertura inicial de todo processo geração, tempo para avaliação inicial, a melhor cobertura de todo o processo e sua respectiva geração e o tempo de execução final.

4.3.2 Exemplo de Uso

O nome do arquivo de configuração deve ser “arqconfig.txt”. O arquivo fonte do programa usado para geração e o arquivo de configuração devem estar no mesmo diretório de execução do módulo ValiGen.

Para exemplificação do uso do módulo ValiGen o programa GCD (*greatest common divisor*) foi utilizado. Este programa calcula o máximo divisor comum entre três números inteiros. Os três números inteiros são usados como entrada, via argumento, e como saída obtém-se o máximo divisor comum entre eles.

Para esse exemplo, o critério escolhido foi Todos-Usos-S. Como o programa GCD possui três números inteiros, o parâmetro formato de entrada foi “III”. O parâmetro variação de

inteiro foi configura de 0/1000. As taxas de *crossover* e mutação foram configuradas com 0,9 e 0,5, respetivamente. O tamanho da população usada pelo AG foi de 50 indivíduos e o número máximo de gerações foi definido como 100. Foram selecionados para a próxima geração, 30 indivíduos por *fitness*, 10 por Elitismo e 10 por Similaridade. Para esta execução, o recurso de lista Tabu foi ativado e o repositório deverá ser consultado por no máximo 90 gerações das 100 configuradas.

O arquivo de configuração utilizado para esta execução é apresentado na Figura 4.4, da Subseção 4.3, no qual é possível observar todos os valores dos parâmetros utilizados para esta execução.

Depois de elaborado o aquivo de configuração, basta executar o módulo com o comando “java -jar ValiGen.jar” no diretório onde se encontra o código fonte do programa a ser testado.

O resultado será o arquivo “População.res” que conterà os 50 dados de teste da melhor geração e o arquivo “Tabu.res” com os melhores dados de teste de todo o processo de geração.

4.4 Considerações Finais

Neste capítulo foi apresentada a abordagem proposta para geração de dados de teste para programas MPI. As estratégias utilizadas para geração dos dados de teste e o tratamento do não determinismo, inerente à atividade de teste de programas concorrentes, também foram abordadas. Um módulo, integrado a ValiMPI, para a abordagem proposta foi apresentado juntamente com um exemplo de uso.

O capítulo a seguir apresenta um estudo experimental que avalia a abordagem proposta.

Avaliação Experimental da Abordagem Proposta

5.1 Considerações Iniciais

Durante a execução de um experimento, diferentes tarefas são realizadas. Essas tarefas estão estabelecidas em diferentes fases. Existem na literatura cinco fases que sempre estarão presente durante o processo de execução de um experimento: definição, planejamento, operação, análise e interpretação dos resultados e, por último, apresentação e empacotamento (Wohlin et al., 2000).

O processo experimental tem início com a fase de definição, na qual o problema é exposto na forma de objetivos e metas do estudo. Com a definição estabelecida dá-se início à segunda fase. Na fase do planejamento, o projeto do experimento é mais detalhado, na qual são apresentadas as hipóteses, variáveis, participantes, instrumentação e aspectos relacionados à validade do experimento são discutidos. Na fase de operação, ocorrem três atividades: preparação, execução e validação dos dados; na preparação o material que será utilizado no experimento é organizado; durante execução ocorre a condução do experimento; a validação dos dados se faz necessária para saber se estes foram coletados corretamente. Após a fase operação é realizada a extração dos dados, que serão analisados e avaliados de uma perspectiva estatística, na fase análise e interpretação dos resultados. Na última fase, apresentação e empacotamento, os resultados obtidos no experimento são apresentados e empacotados por meio de documentos (artigo de pesquisa), pacotes de laboratório ou ainda formando uma base de conhecimento.

Neste capítulo a abordagem para geração de dados apresentada no Capítulo 4 foi avaliada por meio de um estudo experimental, o qual foi planejado e executado com

base no processo de experimentação definido por Wohlin et al. (2000). Para efeito de comparação, a abordagem de geração de dados aleatória foi utilizada.

O capítulo está organizado da seguinte forma: Na Seção 5.2 é apresentada a definição do experimento na qual o objetivo é determinado e questões como o objeto de estudo, o foco na qualidade e o contexto que o estudo será realizado, são apresentadas. Na Seção 5.3 é apresentado o planejamento do experimento, onde são apresentadas as questões de pesquisa e hipóteses e as variáveis envolvidas no experimento. Na Seção 5.4 é apresentado o material de suporte à execução do experimento, como programas e ferramentas utilizadas. Ainda nessa seção os resultados da execução do experimento são apresentados. Na Seção 5.5 é apresentado a análise dos resultados do estudo realizado. Finalmente, na Seção 5.6 são apresentadas as considerações finais deste capítulo.

5.2 Definição do Experimento

O processo experimental tem início com a fase de definição, na qual o problema é exposto na forma de objetivos e metas do estudo. Nesta fase são apresentados o objetivo do experimento e sua forma de condução.

5.2.1 Objetivo

O Objetivo do estudo é avaliar a eficácia em revelar defeitos, a cobertura alcançada e o custo de execução da ferramenta descrita no Capítulo 4, considerando os critérios de teste: todos-nós-s (**tns**), todos-nós-r (**tnr**), todas-arestas-s (**tas**) e todos-s-usos (**tsu**). Para efeito de comparação, a abordagem de geração de dados aleatória foi utilizada.

Com base no objetivo do estudo experimental, as seguintes questões de pesquisa e hipóteses foram definidas:

1. QP1: Qual o percentual de cobertura obtido pelos dados de teste gerados pela ferramenta?
2. QP2: Qual a eficácia em revelar defeitos dos dados de teste gerados pela ferramenta?
3. QP3: Qual o custo de aplicação do algoritmo usado para geração dos dados?

5.2.2 Metas

- **Objeto de estudo:** dados gerados a partir da ferramenta proposta;
- **Propósito:** avaliar a eficácia e o custo dos dados gerados;
- **Foco na qualidade:** custo e eficácia;
- **Perspectiva:** ponto de vista dos pesquisadores;

- **Contexto:** este estudo experimental foi executado por um aluno de mestrado em que os dados gerados, a partir da ferramenta proposta, foram executados sobre um conjunto de programas.

5.3 Planejamento

Com a definição estabelecida dá-se início à segunda fase que é o planejamento. Nesta etapa do experimento é definido e detalhado o contexto, as hipóteses, as variáveis, os participantes, a instrumentação e a validade do estudo. Após a definição do planejamento o estudo experimental estava elaborado e pronto para a execução.

5.3.1 Seleção do Contexto

O estudo foi realizado pelo pesquisador (mestrando) que o definiu. Dois programas implementados em MPI foram selecionados. Esses programas representam programas típicos de programação concorrente e apresentam as características necessárias para a geração de dados automática:

- GCD (*greatest common divisor*) que calcula o máximo divisor comum entre três número inteiros.
- Multiplicação de matrizes (*Mmult*) que multiplica duas matrizes por meio da decomposição de domínios.

Para a análise da eficácia, foram considerados os defeitos inseridos por Brito et al. (2013), os quais representam defeitos típicos em programas concorrentes. Os defeitos inseridos nos programas podem ser visualizados no Figura 5.1.

5.3.2 Hipóteses

QP1: Qual o percentual de cobertura obtido pelos dados de teste gerados pela ValiGen?

Hipótese Nula (H_01): os dados gerados pela ValiGen têm percentual de cobertura equivalente à técnica de Geração Aleatória.

Hipótese Alternativa (H_11): os dados gerados pela ValiGen têm percentual de cobertura superior à técnica de Geração Aleatória.

Hipótese Alternativa (H_21): os dados gerados pela ValiGen têm percentual de cobertura inferior à técnica de Geração Aleatória.

QP2: Qual a eficácia em revelar defeitos, em programas concorrentes, dos dados gerados?

Hipótese Nula (H_02): os dados gerados pela ValiGen revelam quantidade de defeitos equivalente à técnica de Geração Aleatória.

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_Send por MPI_ANY_SOURCE
4	Acessa o índice do 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 parâmetros 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 à variável usando troca de operador
13	Defeito na atribuição à 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 ou MPI_Recv

Figura 5.1: Tipos de defeitos inseridos nos programas

Hipótese Alternativa (H_12): os dados gerados pela ValiGen revelam quantidades de defeitos superior à técnica de Geração Aleatória.

Hipótese Alternativa (H_22): os dados gerados pela ValiGen revelam quantidades de defeitos inferior à técnica de Geração Aleatória.

QP3: Qual o custo dos dados de teste gerados pela ValiGen?

Hipótese Nula (H_03): os dados gerados pela ValiGen têm custo equivalente à técnica de Geração Aleatória.

Hipótese Alternativa (H_13): os dados gerados pela ValiGen têm custo superior à técnica de Geração Aleatória.

Hipótese Alternativa (H_23): os dados gerados pela ValiGen têm custo inferior à técnica de Geração Aleatória.

5.3.3 Seleção dos Indivíduos

O experimento foi conduzido pelo pesquisador que o definiu que cursou disciplinas, como, Engenharia de Software Experimental e Revisão Sistemática, com intuito de agregar a essa pesquisa.

5.3.4 Variáveis

- **Variáveis independentes:**

1. Programas concorrentes utilizados;
2. Defeitos inseridos nos programas;
3. Critérios de teste utilizado.

- **Variáveis dependentes:**

1. Dados gerados pela ValiGen e Geração Aleatória;
2. Custo de execução do algoritmo implementado;
3. Quantidade de defeitos revelados pelos conjuntos de teste;
4. Cobertura obtida pelos conjuntos de teste em relação aos critérios de teste considerados.

5.3.5 Descrição da Instrumentação do Experimento

O experimento tem como objetivo a avaliação dos dados gerados pela ferramenta proposta e o custo, em função do tempo, do algoritmo de geração implementado. Para a execução do experimento foram gerados dados para cada programa usando a abordagem ValiGen e a Geração Aleatória (GA). A fim de se obter relevância na análise estatística, cada abordagem e suas medidas de avaliação, cobertura, eficácia e tempo de execução, foi executada 30 vezes para cada critério de teste e a média dessas execuções foram consideradas.

Inicialmente foram gerados os dados para cada programa usando-se as duas abordagens, obtendo-se assim a cobertura. Os dados gerados pelas duas abordagens eram então armazenados para uso posterior. Ao mesmo tempo em que ocorria a geração dos dados o custo de execução era medido em milissegundos para ambas as abordagens. Para a medição da eficácia, foi utilizada a quantidade de defeitos revelados pelos dados de testes gerados, pelas duas abordagens, em relação a um critério C_1 em função dos defeitos inseridos nos programas.

Diferentes tamanhos de população (conjuntos de dados de teste) foram considerados de modo a avaliar o impacto de cada uma, executando-as com os critérios todos-nós-s (**tns**), todos-nós-r (**tnr**), todas-arestas-s (**tas**) e todos-s-usos (**tsu**). Assim, para cada critério de teste foram consideradas populações contendo 10, 50 e 100 dados de teste. Os parâmetros do AG, da ValiGen, foram os mesmo para todas as configurações e foi configurado da seguinte forma:

- número de gerações = 100;
- taxa *crossover* = 0,9;
- taxa de mutação = 0,5.

Vale ressaltar que antes da escolha definitiva de cada parâmetro do AG, foram feitas várias execuções para o balanceamento dos dos mesmos, sendo os valores supracitados os mais adequados para o problema em questão. Exemplos de configurações utilizadas neste experimento podem ser vistos na Tabela 5.1. Assim, o espaço de execução compreende 2 programas, 4 critérios de teste e 3 conjuntos de teste e para cada um deles, 30 execuções foram realizadas. Isso foi feito para as duas abordagens ValiGen e GA.

Tabela 5.1: Configurações para o experimento

Configuração A	Configuração B
//Aleatória	//Algoritmo Genético
#Criterio = Todos os Nos S	#Criterio = Todos os Nos R
#NumeroArgumentos = 3;	#NumeroArgumentos = 3;
#FormatoEntrada = III;	#FormatoEntrada = III;
#ArquivoFonte = gcd.c;	#ArquivoFonte = gcd.c;
#NomeFuncao = gcd;	#NomeFuncao = gcd;
#CoberturaCriterio = -1;	#CoberturaCriterio = -1;
#VariacaoInt = 1/1000;	#VariacaoInt = 1/1000;
#TaxaCrossover = 0.0;	#TaxaCrossover = 0.9;
#TaxaMutacao = 0.0;	#TaxaMutacao = 0.5;
#TamanhoPopulacao = 10;	#TamanhoPopulacao = 100;
#NumeroGeracoes = 0;	#NumeroGeracoes = 100;
#Elitismo = 0;	#Elitismo = 20;
#Ineditismo = 0;	#Ineditismo = 20;
#Fitness = 0;	#Fitness = 60;
#AtivaTabu = 0;	#AtivaTabu = 1;
#GeracoesComRepositorio=0;	#GeracoesComRepositorio = 90;
#NumeroProcessos = 4;	#NumeroProcessos = 4;
#Funcoes = Master(0)Slave(1,2,3)";	#Funcoes = "Master(0)Slave(1, 2,3)";

5.3.6 Ameaça à Validade

O experimento apresenta ameaça à validade externa, a qual define condições que limitem a habilidade de generalizar os resultados de um experimento. Isso ocorre porque foram empregados somente dois programas e os resultados podem não ser os mesmos para outros programas, o que ocorre com a maioria dos experimentos realizados. Apesar disso, os programas utilizados são representativos, pois apresentam as características mais comuns presentes em aplicações concorrentes com passagem de mensagens. Além disso, diferentes cenários de execução (variando as configurações) foram utilizados de modo a minimizar os impactos dessa ameaça.

5.4 Preparação e Execução

Para execução do experimento foi necessária a realização de algumas tarefas, como a preparação do material de suporte à execução e dos softwares utilizados.

5.4.1 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, já contendo o módulo ValiGen incorporado a mesma;
- Especificação e código fonte dos programas analisados: *GCD*, *Mmult*;
- Especificação e código fonte dos programas analisados: *GCD*, *Mmult*, com defeitos inseridos por Brito et al. (2013).

O ambiente construído para a avaliação dos dados de teste gerados e medição do tempo de execução dos algoritmos possuem as seguintes configurações:

- Notebook: HP EliteBook. Processador: Intel®Core™i7-2630QM, 2.00GHz. Memória: 16Gb;
- Sistema Operacional Linux, distribuição Ubuntu - versão 10.04 (kernel 2.6.32), 32 bits, 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 MPI, oriunda do trabalho de Hausen et al. (2007);
- Open MPI 1.4.
- Java JDK 1.6.0_55
- *Scripts* utilizados para aferição do tempo e eficácia

5.4.2 Descrição dos Programas

Os programas utilizados neste experimento estão implementados na linguagem C e utilizam o padrão MPI de passagem de mensagens. O arquivo fonte desses programas estão disponíveis no Apêndice A. Os programas utilizados foram executados usando quatro processos. A Tabela 5.2 apresenta a quantidade em linhas de código (LOC), a quantidade de pares de funções de troca de mensagens existentes em cada programa (funções MPI Send()/MPI Recv()) e quantidade de elementos requeridos e número de elementos não executáveis (Elem. Req. / Não Exec.). Essas informações fornecem uma ideia do grau de complexidade dos programas.

Tabela 5.2: Característica dos programas utilizados no experimento.

Programa	LOC	Send/Recv	Elem. Req. / Não Exec.			
			tns	tnr	tas	tsu
<i>GCD</i>	112	5/5	7/0	7/0	14/4	34/17
<i>Mmult</i>	192	9/15	15/0	27/0	189/144	252/204

5.4.3 Execução

Nesta etapa o experimento foi executado de acordo com o planejamento elaborado nas fases anteriores.

Para avaliação da cobertura, os dados gerados foram executados na ferramenta ValiMPI e somente os dados com melhor cobertura, para as duas abordagens, foram considerados. As Tabelas 5.3 e 5.4 apresentam a média de cobertura alcançada por ambas abordagens para os programas *GCD* e *Mmult*, respectivamente. Nelas, foram destacados (em negrito) os melhores resultados para cada critério de cada programa. Os elementos não executáveis foram identificados, logo a cobertura máxima é 100%.

Tabela 5.3: Média da cobertura do programa *GCD* em (%).

GCD												
	tns			tnr			tas			tsu		
Tam. Pop.	10	50	100	10	50	100	10	50	100	10	50	100
Aleatória	98,5	100	100	97,6	100	100	98,6	100	100	84,9	88,7	89,4
ValiGen	100	100	100	100	100	100	100	100	100	92,4	100	100

Tabela 5.4: Média da cobertura do programa *Mmult* em (%).

Mmult												
	tns			tnr			tas			tsu		
Tam. Pop.	10	50	100	10	50	100	10	50	100	10	50	100
Aleatória	98,5	100	100	100	100	100	98,6	100	100	100	100	100
ValiGen	100	100	100	100	100	100	100	100	100	100	100	100

Para a análise da eficácia, foram considerados os defeitos inseridos por Brito (Brito et al., 2013), os quais representam defeitos típicos em programas concorrentes. Um dos problemas durante a execução é garantir que um dado de teste execute o conjunto de sincronizações válidas para ele. Quando defeitos são inseridos, devido ao não determinismo, o defeito pode ou não ser revelado. Assim, existem algumas alternativas para execução determinística de programas concorrentes. Neste trabalho, foi utilizada uma técnica simples, a qual executa repetidas vezes o programa na expectativa que outras sincronizações ocorram. É uma técnica de custo baixo que apresenta bons resultados e, portanto, foi empregada neste estudo. A eficácia de cada conjunto de testes é calculada por $\frac{defr}{ndef}$, na qual *defr* é o número de defeitos revelados pelo conjunto e *ndef* é a quantidade total de defeitos. As Tabelas 5.5 e 5.6 apresentam os resultados obtidos em relação à eficácia em revelar defeitos (melhores resultados em negrito) para os programas *GCD* e *Mmult*, respectivamente.

Tabela 5.5: Média da eficácia do programa *GCD* em (%).

GCD												
	tns			tnr			tas			tsu		
Tam. Pop.	10	50	100	10	50	100	10	50	100	10	50	100
Aleatória	52,83	59,44	54,55	57,27	61,72	54,17	56,88	52,44	53,00	52,83	59,44	54,55
ValiGen	58,27	68,22	63,88	55,77	59,33	54,94	55,05	54,27	56,94	58,27	65,44	60,72

Tabela 5.6: Média da eficácia do programa *Mmult* em (%).

Mmult												
	tns			tnr			tas			tsu		
Tam. Pop.	10	50	100	10	50	100	10	50	100	10	50	100
Aleatória	77,03	78,22	77,48	77,25	78,96	77,85	76,54	77,27	77,48	76,96	77,50	76,96
ValiGen	77,77	78,66	80,66	77,93	77,33	78,14	77,57	78,26	80,86	76,81	77,34	77,79

O custo de execução de cada abordagem foi medido em milissegundos. As Tabelas 5.7 e 5.8 apresentam a média de tempo de execução das duas abordagens. A fim de obter melhor visualização comparativa dos valores, a unidade de tempo adotada foi de 33730,4667 milissegundos, que é a menor média de tempo obtida (critério **tns** do programa *GCD* com a técnica aleatória com configuração 10).

Tabela 5.7: Média do tempo de execução do programa *GCD*.

GCD												
	tns			tnr			tas			tsu		
Tam. Pop.	10	50	100	10	50	100	10	50	100	10	50	100
Aleatória	1,00	4,95	9,90	1,02	4,95	9,91	1,07	5,35	10,71	1,14	5,90	11,90
ValiGen	1,15	4,96	9,93	1,12	4,96	9,90	49,46	296,38	617,59	54,32	178,53	613,97

Tabela 5.8: Média do tempo de execução do programa *Mmult*.

Mmult												
	tns			tnr			tas			tsu		
Tam. Pop.	10	50	100	10	50	100	10	50	100	10	50	100
Aleatória	1,93	9,31	18,77	2,20	10,40	20,90	2,19	11,51	21,91	5,54	28,51	59,12
ValiGen	1,85	9,63	19,23	2,11	10,57	17,41	61,48	184,23	410,96	156,65	556,37	914,91

5.5 Análise dos Resultados

Após a fase de preparação e execução do experimento, os dados foram extraídos para análise e interpretação. A seguir são analisados os dados referentes às métricas percentual de cobertura, eficácia em revelar defeitos e custo de execução.

5.5.1 Análise do percentual de cobertura

Na Tabela 5.3, referente ao programa *GCD*, é possível observar que a ValiGen atingiu o máximo de cobertura já com a configuração 10, com exceção do critério **tsu**, no qual o máximo de cobertura foi alcançado com configuração 50. Nos demais cenários a cobertura foi equivalente para as duas técnicas. Este cenário pode ser observado no gráfico de barras na Figura 5.2.

Figura 5.2: Representação gráfica da cobertura do programa *GCD*.

Na Tabela 5.4, referente ao programa *Mmult*, é possível observar que a ValiGen obteve cobertura superior a técnica aleatória, para os critérios **tns** e **tas** com a configuração 10. Este cenário pode ser observado no gráfico de barras na Figura 5.3. Nos demais casos a cobertura foi equivalente para as duas técnicas.

Figura 5.3: Representação gráfica da cobertura do programa *Mmult*.

Assim, é possível observar que a ValiGen atingiu a cobertura máxima possível com a configuração 10 para os dois programas para a maioria dos critérios de teste.

5.5.2 Análise da eficácia

Na Tabela 5.5, referente aos defeitos revelados no programa *GCD*, nota-se que para a maioria dos cenários a ValiGen obteve melhor eficácia, exceto para critério **tnr** a Geração Aleatória obteve melhor resultado. Este cenário pode ser observado no gráfico de barras na Figura 5.4.

Na Tabela 5.6, referente aos defeitos revelados no programa *Mmult*, a ValiGen também obteve melhor desempenho para a maioria dos cenários. Outra semelhança, é que para este programa, o critério **tnr** a Geração Aleatória também obteve melhor resultado. Este cenário pode ser observado no gráfico de barras na Figura 5.5.

É possível observar que a abordagem ValiGen obteve melhores resultados na maioria dos casos apresentados, indicando que a geração de dados usando essa abordagem é capaz

Figura 5.4: Representação gráfica da eficácia do programa *GCD*.

Figura 5.5: Representação gráfica da eficácia do programa *Mmult*.

de gerar dados de teste com alta capacidade de revelar defeitos. É importante observar também que as maiores médias alcançadas estão nos cenários de configuração 50 e 100.

5.5.3 Análise do custo em relação ao tempo de execução

Conforme o esperado, a Geração Aleatória apresenta menor tempo de execução (Tabelas 5.8 e 5.8) para a maioria dos casos avaliados. Porém, pode-se observar que para o programa *Mmult* o tempo de execução para os critérios **tns** e **tnr** foram menores, indicando que nem sempre a abordagem AG apresenta um alto custo. As Figuras 5.6 e 5.7 apresentam o gráfico de barras para o desempenho do custo de execução de cada abordagem para os programas *GCD* e *MMult*.

Pode-se observar também que, apesar da Geração Aleatória apresentar um menor tempo de execução, existiram vários casos que a cobertura da técnica AG com configuração 10 é equivalente a cobertura da técnica aleatória com configuração 50. Por exemplo, para o *GCD*, com relação ao critério **tns**, a ValiGen obteve cobertura de 100% e tempo de 1,15 na configuração 10 e a Geração Aleatória obteve 100% apenas com configuração 50 e seu tempo de execução foi de 4,95. Assim sendo, ao comparar os tempos das duas

Figura 5.6: Representação gráfica do custo de execução no programa *GCD*.

Figura 5.7: Representação gráfica do custo de execução no programa *Mmult*.

configurações, o AG mostra-se mais vantajoso por obter melhor cobertura em menor tempo.

5.5.4 Análise das Hipóteses

Para análise das hipóteses um teste estatístico foi realizado. Esse teste indica se existem evidências estatísticas significativas para os resultados obtidos nas médias alcançadas para os critérios de avaliação cobertura, eficácia e tempo de execução pelas duas abordagens. Assim, o teste T foi considerado.

Para a análise estatística foi admitido o nível de significância de 95%, assim sendo, essa evidência se dá quando o valor obtido para o *Teste t* é inferior a 0,05, onde nesse caso a H_0 pode ser rejeitada. As Tabelas 5.9 e 5.10 apresentam os resultados obtidos pelo teste T.

Com relação à cobertura, analisando as médias de cobertura (Tabelas 5.3, 5.4) da perspectiva estatística, nos casos em que a ValiGen foi melhor que a aleatória, somente nos cenários de configuração 10 dos critérios **tns**, dos dois programas, e **tas** do programa *GCD*, não foi possível comprovar as diferenças das médias, ou seja, o valor de Significância (Tabelas 5.10 e 5.9) foi superior a 0,05. Nos outros casos de diferenças nas médias a

Tabela 5.9: Teste T para o programa *GCD*.

GCD												
	tns			tnr			tas			tsu		
Tam. Pop.	10	50	100	10	50	100	10	50	100	10	50	100
Cobertura	,083	*	*	,023	*	*	,083	,000	,000	,000	,000	,000
Eficácia	,060	,003	,001	,731	,310	,580	,358	,227	,036	,000	,012	,007
Tempo	,000	,044	,001	,268	,300	,660	,000	,000	,000	,000	,000	,000

* T não pode ser calculado porque o desvio padrão de ambos os grupos é 0

Tabela 5.10: Teste T para o programa *Mmult*.

Mmult												
tns				tnr			tas			tsu		
Tam. Pop.	10	50	100	10	50	100	10	50	100	10	50	100
Cobertura	,083	*	*	,023	*	*	*	*	*	*	*	*
Eficácia	,071	,408	,001	,045	,081	,521	,006	,003	,001	,708	,707	,060
Tempo	,006	,002	,000	,001	,000	,001	,000	,000	,000	,000	,000	,000

* T não pode ser calculado porque o desvio padrão de ambos os grupos é 0.

hipótese H_{11} foi aceita. No geral, os resultados estatísticos para a cobertura indicam que a hipótese H_{01} foi aceita, ou seja, que a média de cobertura de ambas as abordagens são equivalentes.

Para a eficácia, (Tabelas 5.5, 5.6), na maioria dos casos em que o ValiGen se sobressaiu, a hipótese H_{02} foi rejeitada, logo estatisticamente a ValiGen revela mais defeitos que a aleatória, com exceção do critério **tsu** do programa *Mmult*. Para os casos de melhor desempenho da técnica aleatória, a hipótese H_{02} foi aceita, logo a diferença entre as médias não foi comprovada estatisticamente.

Para a análise da média de tempo de execução (Tabelas 5.7 e 5.8) apenas para o critério **tnr**, em todas as configurações, do programa *GCD* e **tns** do programa *Mmult*, casos de diferenças de médias, a hipótese H_{03} foi aceita, indicando que a abordagem as duas abordagens possui tempo equivalente. Nos outros cenários, casos que a média de custo da ValiGen é maior, o nível de significância estatística foi sempre superior 0,05 indicando que esta abordagem, no geral, apresenta um maior custo.

5.6 Considerações Finais

Este capítulo apresentou a definição, o planejamento a execução e análise dos dados de um estudo experimental baseado no processo de experimentação definido por (Wohlin et al., 2000). O objetivo do estudo realizado era a avaliação da cobertura, eficácia em revelar defeitos e custo do tempo de execução da abordagem proposta no Capítulo 4.

No capítulo a seguir é apresentada a conclusão dessa dissertação de mestrado.

Conclusão e Trabalhos Futuros

6.1 Caracterização da Pesquisa Realizada

Os programas concorrentes possuem características que os diferem de programas sequenciais, como comunicação, sincronização, paralelismo e concorrência, que precisam ser consideradas durante a atividade de teste. Essas características também apresentam-se como desafio durante a atividade de teste motivando o desenvolvimento de novas técnicas de testes ou adaptando técnicas usadas em programas sequenciais para esse contexto.

A atividade de teste é apontada como uma das mais onerosas no desenvolvimento de software (Harrold, 2000). Um relatório preparado para o NIST (*National Institute of Standards and Technology*) quantifica os altos impactos econômicos de uma infraestrutura de teste de software inadequada e como melhoria cita, por exemplo, a geração automática de teste (Tassey, 2002).

As técnicas de meta-heurística tem sido objeto de estudo entre os pesquisadores para proposição de técnicas de geração automática de dados de teste, pois apresentam soluções para problemas indecidíveis a uma complexidade computacional e tempo aceitável.

Este trabalho apresentou uma abordagem para geração automática de dados de teste que utiliza Algoritmo Genético, a qual é capaz de gerar dados que cobrem os requisitos de teste dos critérios estruturais propostos para programas concorrentes que usam o padrão passagem de mensagem MPI. Um estudo experimental foi realizado a fim de avaliar a abordagem proposta, analisando a cobertura obtida, a eficácia em revelar defeitos e o custo de execução da abordagem de geração comparada com a geração aleatória. Os resultados mostram que a abordagem proposta, para a medida cobertura, foi equivalente em muitos cenários, porém nunca obteve desempenho inferior, e nos cenários que obteve melhor desempenho, a maioria obteve garantia estatisticamente. Em relação à eficácia, a abordagem proposta também obteve melhor desempenho, sendo que, nos cenários em que

a abordagem aleatória se sobressaiu, não houve comprovação estatística. Em relação ao tempo de execução, a técnica aleatória se sobressaiu. Contudo, é importante ressaltar que, apesar da técnica aleatória obter melhor desempenho na medida de tempo, configurações maiores da técnica aleatória obtiveram desempenho similar a configurações menores da ValiGen.

6.2 Contribuições

As principais contribuições deste mestrado são:

- Levantamento sobre as principais abordagens de meta-heurística para a geração de dados de teste;
- Definição de uma abordagem de geração automática de dados para programas concorrentes com passagem de mensagem;
- Extensão das funcionalidades da ferramenta de teste ValiMPI, implementando um módulo para geração de dados de teste, chamado ValiGen, o qual automatiza a abordagem definida;
- Estudo comparativo mostrando que a abordagem proposta é adequada para a geração de dados para programas concorrentes;
- Produção de um artigo científico com os resultados oriundos da pesquisa realizada.

6.3 Dificuldades e Limitações

Durante o desenvolvimento do trabalho algumas dificuldades ocorreram. A primeira dificuldade foi durante a escolha dos programas a serem usados para geração dos dados de teste. Os programas a serem utilizados precisavam estar codificados dentro das limitações da ferramenta de apoio ValiMPI como também da ferramenta de geração dos dados ValiGen. Outra dificuldade encontrada foi o uso da ferramenta de apoio ValiMPI, pois foi necessário encontrar qual versão desta ferramenta dava suporte à implementação da abordagem proposta. Para isso foi necessário o entendimento de alguns *scripts* para saber seu funcionamento para poder fazer a interação com a ferramenta de geração dos dados implementada.

6.4 Trabalhos Futuros

Como trabalhos futuros pretende-se investigar:

- Melhorar na priorização dos dados de teste para a geração de novos dados durante a evolução da abordagem;

- Forma de tratamento dos elementos não executáveis, pois esses interferem no tempo de execução da abordagem ValiGen;
- Desenvolver novos estudos experimentais, considerando mais programas, com outras características e comparando com outras abordagens de geração de dados;
- Estabelecer novas abordagens de geração de dados de teste para programas concorrentes, considerando outras técnicas de meta-heurística existentes na literatura.

Referências

- Almasi, G.; Gottlieb, A. *Highly parallel computing*. The Benjamin/Cummings Series in Computer Science and Engineering. Benjamin/Cummings Pub. Co., 1994.
- Andreou, A. S.; Economides, K. A.; Sofokleous, A. A. An automatic software test-data generation scheme based on data flow criteria and genetic algorithms. In: *Proceedings of the 7th IEEE International Conference on Computer and Information Technology*, Washington, DC, USA, 2007, p. 867–872.
- Andrews, G. *Foundations of multithreaded, parallel, and distributed programming*. Addison-Wesley, 2000.
- Barbosa, E. F.; Chaim, M. L. Vicenzi, A. M. R.; Delamaro, M. E.; Jino, M.; Maldonado, J. C. a. Teste estrutural. In: Delamaro, M. E.; Jino, M.; Maldonado, J. C., eds. *Introdução ao Teste de Software*, Elsevier, p. 47 – 76, 2007.
- Bin, L.; Zhi-shu, L.; Yan-Hong, C.; Bao-lin, L. Automatic test data generation tool based on genetic simulated annealing algorithm. In: *International Conference on Computational Intelligence and Security Workshops, 2007. (CISW 2007)*, 2007, p. 183 –186.
- Bottaci, L. A Genetic Algorithm Fitness Function for Mutation Testing. In: *SEMI-NAL'2001 – First International Workshop on Software Engineering using Metaheuristic INnovative ALgorithms*, Toronto, Ontario, Canada, 2001.
- Boyer, R. S.; Elspas, B.; Levitt, K. N. A formal system for testing and debugging programs by symbolic execution. In: *Proceedings of the international conference on Reliable software*, New York, NY, USA: ACM, 1975, p. 234–245.
- Brito, M. A.; Souza, S. R.; Souza, P. S. An empirical evaluation of the cost and effectiveness of structural testing criteria for concurrent programs. *Procedia Computer Science*, v. 18, n. 0, p. 250 – 259, 2013 International Conference on Computational Science, 2013.
- Bueno, P. M.; Jino, M.; Wong, W. E. Diversity oriented test data generation using metaheuristic search techniques. *Information Sciences*, p. 24 – 44, 2011.

- Bueno, P. M. S.; Jino, M.; Jino, M. Automatic test data generation for program paths using genetic algorithms. In: *SEKE*, 2001, p. 2–9.
- de Castro, L. *Fundamentals of natural computing: Basic concepts, algorithms, and applications*. Chapman & Hall/CRC Computer & Information Science Series. Taylor & Francis, 2006.
- Chaim, L. *POKE-TOOL: Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados*. Dissertação de Mestrado, Dissertacao de Mestrado - DCA/FEEC/UNICAMP, 1991.
- Chen, T. Adaptive random testing. In: *The Eighth International Conference on Quality Software (QSIC 2008)*, 2008, p. 443.
- Chusho, T. Test data selection and quality estimation based on the concept of essential branches for path testing. *IEEE Trans. Softw. Eng.*, v. 13, n. 5, p. 509–517, 1987.
- Dara, R.; Li, S.; Liu, W.; Smith-Ghorbani, A.; Tahvildari, L. Using dynamic execution data to generate test cases. In: *IEEE International Conference on Software Maintenance (ICSM 2009)*, 2009, p. 433–436.
- Davis, L. D.; Mitchell, M. Handbook of genetic algorithms. Van Nostrand Reinhold, 1991.
- De Jong, K. A. *An analysis of the behavior of a class of genetic adaptive systems*. Tese de Doutorado, Ann Arbor, MI, USA, 1975.
- Delamaro, M.; Maldonado, J.; Jino, M. *Introdução ao teste de software*. Elsevier, 2007a.
- Delamaro, M. E. *Proteum: Um ambiente de teste baseado na análise de mutantes*. Dissertação de Mestrado, ICMC-USP, São Carlos, SP, 1993.
- Delamaro, M. E.; Maldonado, J. C.; Jino, M. Conceitos básicos. In: Delamaro, M. E.; Jino, M.; Maldonado, J. C., eds. *Introdução ao Teste de Software*, Elsevier, p. 1 – 7, 2007b.
- DeMillo, R. A.; McCracken, W. M.; Martin, R. J.; Passafiume, J. F. *Software testing and evaluation*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1987.
- 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: *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, New York, NY, USA: ACM, 2005, p. 78–82.
- Dongarra, J.; Foster, I.; Fox, G.; Gropp, W.; Kennedy, K.; Torczon, L.; White, A. *Source book of parallel computing*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Morgan Kaufmann, 2003.
- Duran, J. W.; Ntafos, S. C. An evaluation of random testing. *IEEE Transactions on Software Engineering*, v. SE-10, n. 4, p. 438–444, 1984.

- Eytani, Y. Concurrent java test generation as a search problem. *Electronic Notes in Theoretical Computer Science*, v. 144, n. 4, p. 57 – 72, 2006.
- Fabbri, S. C. P. F.; Vicenzi, A. M. R.; Maldonado, J. C. Teste funcional. In: Delamaro, M. E.; Jino, M.; Maldonado, J. C., eds. *Introdução ao Teste de Software*, Elsevier, p. 9 – 24, 2007.
- Ferguson, R.; Korel, B. Generating test data for distributed software using the chaining approach. *Information and Software Technology*, v. 38, n. 5, p. 343 – 353, 1996.
- Ferreira, L. P.; Vergilio, S. R. Tdsgen: An environment based on hybrid genetic algorithms for generation of test data. In: *SEKE*, 2005, p. 312–317.
- Ghiduk, A.; Harrold, M.; Girgis, M. Using genetic algorithms to aid test-data generation for data-flow coverage. In: *14th Asia-Pacific Software Engineering Conference. (APSEC 2007)*, 2007, p. 41 –48.
- Goldberg, D. E. *Genetic algorithms in search, optimization and machine learning*. 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.
- Goldberg, D. E.; Richardson, J. Genetic algorithms with sharing for multimodal function optimization. In: *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, 1987, p. 41–49.
- Grama, A.; Karypis, G.; Kumar, V.; Gupta, A. *Introduction to Parallel Computing (2nd Edition)*. 2 ed. Addison Wesley, 2003.
- Harman, M.; McMinn, P. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Trans. Softw. Eng.*, v. 36, n. 2, p. 226–247, 2010.
- Harrold, M. J. Testing: a roadmap. In: *Proceedings of the Conference on The Future of Software Engineering (ICSE)*, New York, NY, USA: ACM, 2000, p. 61–72.
- Hausen, A. C.; Vergilio, S. R.; Souza, S. R. S.; Souza, P. S. L.; Simao, A. S. A tool for structural testing of mpi programs. In: *8th IEEE Latin-American Test Workshop, Cuzco, Lima.*, 2007.
- Holland, J. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.
- Hrubá, V.; Křena, B.; Letko, Z.; Ur, S.; Vojnar, T. Testing of concurrent programs using genetic algorithms. In: *Proceedings of the 4th International Conference on Search Based Software Engineering*, 2012, p. 152–167.
- Hwang, K.; Xu, Z. *Scalable parallel computing: Technology, architecture, programming*. Computer engineering series. WCB/McGraw-Hill, 1998.
- IEEE Ieee std 610.12-1990). *IEEE Standard Glossary of Software Engineering Terminology*, 1990.

- Jones, B. F.; Sthamer, H. H.; Eyres, D. E. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, v. 11, n. 5, p. 299–306, 1996.
- Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, v. 220, 4598, p. 671–680, 1983.
- Koirala, S.; Sheikh, S. *Software testing interview questions*. Computer science series. Ebsco Publishing, 2009.
- Korel, B. Automated software test data generation. *IEEE Trans. Softw. Eng.*, v. 16, n. 8, p. 870–879, 1990.
- Korel, B.; Wedde, H.; Ferguson, R. Dynamic method of test data generation for distributed software. *Information and Software Technology*, v. 34, n. 8, p. 523 – 531, 1992.
- Krawczyk, H.; Wiszniewski, B.; Mork, P. Classification of software defects in parallel programs. In: *USENIX Technical Conference*, 1994.
- Krena, B.; Letko, Z.; Vojnar, T. Noise injection heuristics for concurrency testing. In: *MEMICS*, 2011, p. 123–135 (*Lecture Notes in Computer Science*, v.7119).
- Lei, Y.; Carver, R. H. Reachability testing of concurrent programs. *IEEE Trans. Softw. Eng.*, v. 32, n. 6, p. 382–403, 2006.
- Letko, Z. Sophisticated testing of concurrent programs. In: *Proceedings of the 2nd International Symposium on Search Based Software Engineering (SSBSE '10)*, Benevento, Italy, 2010, p. 36–39.
- Lewis, B.; Berg, D. *Threads primer: A guide to multithreaded programming*. SunSoft Press, 1996.
- Lin, J.-C.; Yeh, P.-L. Automatic test data generation for path testing using gas. *Information Sciences*, v. 131, p. 47 – 64, 2001.
- Louzada, J.; Camilo-Junior, C.; Vincenzi, A.; Rodrigues, C. An elitist evolutionary algorithm for automatically generating test data. In: *IEEE Congress on Evolutionary Computation (CEC 2012)*, 2012, p. 1 –8.
- McMinn, P. Search-based software test data generation: A survey. *Software Testing Verification Reliability*, v. 14, n. 2, p. 105–156, 2004.
- Michael, C. C.; McGraw, G.; Schatz, M. A. Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, v. 27, n. 12, p. 1085–1110, 2001.
- Michalewicz, Z.; Fogel, D. B. *How to solve it: Modern heuristics*. Enlarged 2nd ed. Springer, 2004.
- Miller, W.; Spooner, D. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, v. SE-2, n. 3, p. 223 – 226, 1976.

- Mitchell, M. *An introduction to genetic algorithms*. Cambridge, MA, USA: MIT Press, 1998.
- Myers, G. J. *The art of software testing*. 2 ed. John Wiley Sons, Inc., 2004.
- Offutt, A.; Seaman, E. Using symbolic execution to aid automatic test data generation. In: *Fifth Annual Conference on Computer Assurance (COMPASS 1990)*, 1990, p. 12–21.
- Ombuki, B. M.; Nakamura, M.; Osamu, M.; Ombuki, B. M. A hybrid search based on genetic algorithms and tabu search for vehicle routing. In: *Routing, 6th IASTED Intl. Conf. On Artificial Intelligence and Soft Computing (ASC 2002)*, ACTA Press, 2002, p. 17618–1.
- Pacheco, P. S. *Parallel programming with mpi*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.
- Pargas, R. P.; Harrold, M. J.; Peck, R. R. Test-data generation using genetic algorithms. *Software Testing, Verification And Reliability*, v. 9, p. 263–282, 1999.
- Patton, R. *Software testing (2nd edition)*. Indianapolis, IN, USA: Sams, 2005.
- Pinto, G. H. L.; Vergilio, S. R. A multi-objective genetic algorithm to test data generation. In: *22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2010)*, 2010, p. 129–134.
- Pressman, R. *Software engineering: A practitioner's approach*. McGraw-Hill higher education. McGraw-Hill Higher Education, 2010.
- Quinn, M. *Parallel programming in c with mpi and openmp*. McGraw-Hill Higher Education. McGraw-Hill, 2004.
- Rapps, S.; Weyuker, E. J. Data flow analysis techniques or test data selection. In: *Proceedings of the 6th international conference on Software engineering, ICSE '82*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, p. 272–278 (*ICSE '82*,).
- Rapps, S.; Weyuker, E. J. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, v. 11, n. 4, p. 367–375, 1985.
- Rayward-Smith, V. *Modern heuristic search methods*. Wiley, 1996.
- Russell, S. J.; Norvig, P. *Artificial intelligence: A modern approach*. 3rd ed. Prentice Hall, 2009.
- Sarmanho, F. S.; Souza, P. S.; Souza, S. R.; Simao, A. S. Structural testing for semaphore-based multithread programs. In: *Proceedings of the 8th international conference on Computational Science, Part I, ICCS '08*, Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–346 (*ICCS '08*,).
- Simao, A. S.; Souza, S. R.; Maldonado, J. C. A family of coverage testing criteria for coloured petri nets. *XVII Simposio Brasileiro de Engenharia de Software*, p. 209 – 224, 2003.

- Soares, A. K. E. M. S. *Qualidade de software: aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software*. 2 ed. NovaTec Editora, 2007.
- Souza, P. L.; Sawabe, E. T.; Silva Simao, A.; Vergilio, S. R.; Rocio Senger De Souza, S. ValiPVM - a graphical tool for structural testing of pvm programs. In: *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Berlin, Heidelberg: Springer-Verlag, 2008a, p. 257–264.
- Souza, S.; Paulo; Machado, M.; Camillo, M.; Simão, A.; Zaluska, E. Using coverage and reachability testing to improve concurrent program testing quality. In: *23rd International Conference on Software Engineering and Knowledge Engineering*, 2011, p. 207–212.
- Souza, S. R. S.; Vergilio, S. R.; Souza, P. S. L.; Simao, A. S.; Hausen, A. C. Structural testing criteria for message-passing parallel programs. *Concurrency and Computation: Practice and Experience*, v. 20, n. 16, p. 1893–1916, 2008b.
- Souza, S. R. S.; Vergilio, S. R.; Souza, S. P. L. Teste de programas concorrentes. In: Delamaro, M. E.; Jino, M.; Maldonado, J. C., eds. *Introdução ao Teste de Software*, Elsevier, p. 1 – 7, 2007.
- Tanenbaum, A. *Organização estruturada de computadores*. Pearson Prentice Hall, 2007a.
- Tanenbaum, A. S. *Modern operating systems*. 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007b.
- Tanenbaum, A. S.; Steen, M. v. *Distributed systems: Principles and paradigms (2nd edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.
- Tasse, G. *The economic impacts of inadequate infrastructure for software testing*. Relatório Técnico, 2002.
- Taylor, R. N.; Levine, D. L.; Kelly, C. D. Structural testing of concurrent programs. *IEEE Trans. Softw. Eng.*, v. 18, n. 3, p. 206–215, 1992.
- Tian, T.; Gong, D. Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms. *Chinese Journal of Electronics*, v. 23, n. 2, p. 291 – 296, 2014.
- Vergilio, S. R.; Maldonado, J. C.; Jino, M. Geração de dados de teste. In: Delamaro, M. E.; Jino, M.; Maldonado, J. C., eds. *Introdução ao Teste de Software*, Elsevier, p. 269 – 291, 2007.
- Vergilio, S. R.; Souza, S. R. S.; Souza, P. S. L. Coverage testing criteria for message-passing parallel programs. In: *IEEE 6th IEEE Latin American Test Workshop (LATW 2005)*, 2005, p. 161–166.

- 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.; Lei, Y.; Ma, X. Effective generation of test sequences for structural testing of concurrent programs. In: *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005)*., 2005, p. 539 – 548.
- Yang, C. D.; Souter, A. L.; Pollock, L. L. All-du-path coverage for parallel programs. In: *ACM SigSoft International Symposium on Software Testing and Analysis*, ACM Press, 1998, p. 153–162.
- Yang, C. S. D.; Pollock, L. All-uses testing of shared memory parallel programs. *Software Testing, Verification and Reliability (STVR)*, v. 13, p. 3–24, 2003.
- Zhang, B.; Wang, C. Automatic generation of test data for path testing by adaptive genetic simulated annealing algorithm. In: *IEEE International Conference on Computer Science and Automation Engineering (CSAE 2011)*, 2011, p. 38 –42.
- Zhang, Y.; Gong, D. Evolutionary generation of test data for multiple paths coverage with faults detection. In: *IEEE 5th International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA 2010)*, 2010, p. 406 –410.

Apêndice

A.1 Arquivo fonte do programa GCD

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4 #include "gcd.h"
5
6 void Master(int, int, int);
7 void Slave(int rank);
8
9 int main(int argc, char** argv)
10 {
11     int myRank;
12     int x, y, z;
13
14     if (argc != 4)
15     {
16         puts("argc != 4");
17         return 1;
18     }
19     MPI_Init(&argc, &argv);
20
21     MPI_Comm_rank(MPLCOMM_WORLD, &myRank);
22
23     if (myRank == 0)
24     {
25         x = atoi(argv[1]);
26         y = atoi(argv[2]);
27         z = atoi(argv[3]);
28         Master(x, y, z);
29     }
```

```

30     else
31     {
32         Slave(myRank);
33     }
34     MPI_Finalize();
35     return 0;
36 }
37
38
39 //Processo Mestre
40 void Master(int x, int y, int z)
41 {
42     int buf[3];
43
44     // envia x e y
45     buf[0] = x;
46     buf[1] = y;
47     MPI_Send(buf, 2, MPLINT, 1, 1, MPLCOMM_WORLD);
48
49     // envia y e z
50     buf[0] = y;
51     buf[1] = z;
52     MPI_Send(buf, 2, MPLINT, 2, 1, MPLCOMM_WORLD);
53
54     MPI_Recv(buf, 1, MPLINT, MPLANY_SOURCE, 2, MPLCOMM_WORLD, &status);
55     x = buf[0];
56
57     MPI_Recv(buf, 1, MPLINT, MPLANY_SOURCE, 2, MPLCOMM_WORLD, &status);
58     y = buf[0];
59
60     if (x > 1 && y > 1)
61     {
62         buf[0] = x;
63         buf[1] = y;
64         MPI_Send(buf, 2, MPLINT, 3, 1, MPLCOMM_WORLD);
65         MPI_Recv(buf, 1, MPLINT, MPLANY_SOURCE, 2, MPLCOMM_WORLD,
66 &status);
67         z = buf[0];
68     }
69     else
70     {
71         buf[0] = -1;
72         buf[1] = -1;
73         MPI_Send(buf, 2, MPLINT, 3, 1, MPLCOMM_WORLD);
74         MPI_Recv(buf, 1, MPLINT, 3, 2, MPLCOMM_WORLD, &status);
75         z = 1;
76     }
77     printf("resultado = %d\n", z);
78 }
79
80
81 //Processo Escravo
82 void Slave(int rank)
83 {
84     int buf[2];
85

```

```

86     MPI_Recv(buf, 2, MPLINT, 0, MPLANY_TAG, MPLCOMMWORLD, &status);
87
88     while (buf[0] != buf[1])
89     {
90         if (buf[0] < buf[1])
91             buf[1] = buf[1] - buf[0];
92         else
93             buf[0] = buf[0] - buf[1];
94     }
95     MPI_Send(buf, 1, MPLINT, 0, 2, MPLCOMMWORLD);
96 }

```

gcd.c

A.2 Arquivo fonte do programa Mmult

```

1  #include <stdio.h>
2  #include <mpi.h>
3  #include "mmult.h"
4
5  /* Variables */
6  int numtasks, /* number of tasks in partition */
7      taskid, /* a task identifier */
8      numworkers, /* number of worker tasks */
9      source, /* task id of message source */
10     dest, /* task id of message destination */
11     mtype, /* message type */
12     rows, /* rows of matrix A sent to each worker */
13     averow, extra, offset,
14     i, j, k, /* misc */
15     count,
16     nra, nca, ncb;
17 double a[MAXNRA * MAXNCA], /* matrix A to be multiplied */
18        b[MAXNCA * MAXNCB], /* matrix B to be multiplied */
19        c[MAXNRA * MAXNCB]; /* result matrix C */
20
21 void master(char *fpath)
22 {
23     fp = fopen(fpath, "r");
24     fscanf(fp, "%d %d", &nra, &nca);
25     for (i = 0; i < nra; i++)
26     {
27         for (j = 0; j < nca; j++)
28         {
29             fscanf(fp, "%lf", &a[i * nca + j]);
30         }
31     }
32
33     fscanf(fp, "%d", &ncb);
34     for (i = 0; i < nca; i++)
35     {
36         for (j = 0; j < ncb; j++)

```



```

37     {
38         fscanf(fp, "%lf", &b[i * ncb + j]);
39     }
40 }
41
42 /* send matrix data to the worker tasks */
43 averow = nra / numworkers;
44 extra = nra % numworkers;
45 offset = 0;
46 mtype = FROMMASTER;
47 for (dest = 1; dest <= numworkers; dest++)
48 {
49     if (dest <= extra)
50         rows = averow + 1;
51     else
52         rows = averow;
53     MPLSend(&offset, 1, MPLINT, dest, FROMMASTER, MPLCOMMLWORLD);
54     MPLSend(&rows, 1, MPLINT, dest, FROMMASTER, MPLCOMMLWORLD);
55     MPLSend(&nca, 1, MPLINT, dest, FROMMASTER, MPLCOMMLWORLD);
56     count = rows * nca;
57     MPLSend(&a[nca * offset], count, MPLDOUBLE, dest, FROMMASTER,
58 MPLCOMMLWORLD);
59     count = nca * ncb;
60     MPLSend(&ncb, 1, MPLINT, dest, FROMMASTER, MPLCOMMLWORLD);
61     MPLSend(&b, count, MPLDOUBLE, dest, FROMMASTER,
62 MPLCOMMLWORLD);
63     offset = offset + rows;
64 }
65
66 /* wait for results from the 3 worker tasks */
67 MPLRecv(&offset, 1, MPLINT, MPLANY_SOURCE, MPLANY_TAG,
68 MPLCOMMLWORLD, &status);
69 source = status.MPLSOURCE;
70 MPLRecv(&rows, 1, MPLINT, source, MPLANY_TAG, MPLCOMMLWORLD,
71 &status);
72 count = rows * ncb;
73 MPLRecv(&c[offset * ncb], count, MPLDOUBLE, source, MPLANY_TAG,
74 MPLCOMMLWORLD, &status);
75 MPLRecv(&offset, 1, MPLINT, MPLANY_SOURCE, MPLANY_TAG,
76 MPLCOMMLWORLD, &status);
77 source = status.MPLSOURCE;
78 MPLRecv(&rows, 1, MPLINT, source, MPLANY_TAG, MPLCOMMLWORLD,
79 &status);
80 count = rows * ncb;
81 MPLRecv(&c[offset * ncb], count, MPLDOUBLE, source, MPLANY_TAG,
82 MPLCOMMLWORLD, &status);
83 MPLRecv(&offset, 1, MPLINT, MPLANY_SOURCE, MPLANY_TAG,
84 MPLCOMMLWORLD, &status);
85 source = status.MPLSOURCE;
86 MPLRecv(&rows, 1, MPLINT, source, MPLANY_TAG, MPLCOMMLWORLD,
87 &status);
88 count = rows * ncb;
89 MPLRecv(&c[offset * ncb], count, MPLDOUBLE, source, MPLANY_TAG,
90 MPLCOMMLWORLD, &status);
91
92 /* print results */

```

```

93     printf("Here is the result matrix\n");
94     for (i = 0; i < nra; i++)
95     {
96         printf("\n");
97         for (j = 0; j < ncb; j++)
98             printf("%6.2f", c[i * ncb + j]);
99     }
100    printf("\n");
101 }
102
103 void slave(void)
104 {
105     MPI_Recv(&offset, 1, MPLINT, MASTER, MPLANY_TAG, MPLCOMMLWORLD,
106 &status);
107     MPI_Recv(&rows, 1, MPLINT, MASTER, MPLANY_TAG, MPLCOMMLWORLD,
108 &status);
109     MPI_Recv(&nca, 1, MPLINT, MASTER, MPLANY_TAG, MPLCOMMLWORLD,
110 &status);
111     count = rows * nca;
112     MPI_Recv(&a, count, MPLDOUBLE, MASTER, MPLANY_TAG, MPLCOMMLWORLD,
113 &status);
114     MPI_Recv(&ncb, 1, MPLINT, MASTER, MPLANY_TAG, MPLCOMMLWORLD,
115 &status);
116     count = nca * ncb;
117     MPI_Recv(&b, count, MPLDOUBLE, MASTER, MPLANY_TAG, MPLCOMMLWORLD,
118 &status);
119
120     for (k = 0; k < ncb; k++)
121         for (i = 0; i < rows; i++) {
122             c[i * ncb + k] = 0.0;
123             for (j = 0; j < nca; j++)
124                 c[i * ncb + k] = c[i * ncb + k] + a[i * nca + j]
125 * b[j * ncb + k];
126         }
127
128     /* after computing */
129     MPI_Send(&offset, 1, MPLINT, MASTER, FROMWORKER, MPLCOMMLWORLD);
130     MPI_Send(&rows, 1, MPLINT, MASTER, FROMWORKER, MPLCOMMLWORLD);
131     MPI_Send(&c, rows * ncb, MPLDOUBLE, MASTER, FROMWORKER,
132 MPLCOMMLWORLD);
133     /* after send */
134 }
135
136 int main(int argc, char **argv)
137 {
138
139     MPI_Init(&argc, &argv);
140     MPI_Comm_rank(MPLCOMMLWORLD, &taskid);
141     MPI_Comm_size(MPLCOMMLWORLD, &numtasks);
142     numworkers = numtasks - 1;
143
144     /****** master task *****/
145     if (taskid == MASTER)
146     {
147         master(argv[1]);
148     }
149     /* end of master section */

```

```
149
150  /****** worker task *****/
151  if (taskid > MASTER)
152  {
153      slave();
154  }
155      /* end of worker */
156  MPI_Finalize();
157  return 0;
158 }
```

mmult.c