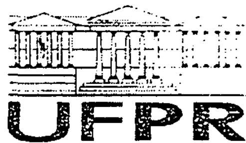
MARIA CLÁUDIA FIGUEIREDO PEREIRA EMER

SELEÇÃO E AVALIAÇÃO DE DADOS DE TESTE BASEADAS EM PROGRAMAÇÃO GENÉTICA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Curso de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora:
Prof.ª Dr.ª Silvia Regina Vergilio

CURITIBA
2002



Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática da aluna *Maria Claudia Figueiredo Pereira Emer*, avaliamos o trabalho intitulado ***"Seleção e Avaliação de Dados de Teste Baseadas em Programação Genética"***, cuja defesa foi realizada no dia 13 de março de 2002, às quinze horas, no anfiteatro A do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação da candidata.

Curitiba, 13 de março de 2002.

Profª. Dra. Silvia Regina Vergilio
DINF/UFPR - Orientadora

Prof. Dr. José Carlos Maldonado
ICMC-USP/São Carlos

Profª. Dra. Aurora Trinidad Ramírez Pozo
DINF/UFPR

// Agradecimentos

Agradeço,

primeiramente a Deus, que está presente em todos os momentos de minha vida;

à minha família, especialmente ao meu marido Ivo, meus pais, Gizelia e Antonio, pela compreensão, paciência, apoio, companheirismo e amor dedicados, também, nas horas mais difíceis;

à minha orientadora, Profª. Silvia, por ser amiga, companheira e incentivadora, tornando a fase de elaboração da dissertação algo mais prazeroso e menos solitário;

à Profª Aurora por sua amizade e auxílio na realização do trabalho;

aos amigos que conheci no mestrado, especialmente àqueles que estiveram mais próximos, Eduardo, Ernesto, Andréa, Celso e Cláudio;

ao Prof. José Carlos Maldonado pela sua contribuição neste trabalho.

Resumo

Programação Genética (PG) tem sido aplicada, recentemente, para resolver problemas em diversas áreas. Tem por objetivo a indução de programas a partir de dados de treinamento, usando conceitos da teoria da evolução de Darwin. Por outro lado, o teste de software, que é uma atividade fundamental e custosa para assegurar a qualidade de software, tem por objetivo a geração de casos de teste a partir de um programa sendo testado. Dessa forma, é vista uma simetria entre a indução de programas baseada em PG e a atividade de teste. Este trabalho explora essa simetria e propõe uma abordagem de teste de software baseada em PG. Critérios de teste baseados em erros, geralmente, derivam dados de teste a partir um conjunto de operadores de mutação para produzir alternativas que diferem do programa em teste por uma simples modificação. A abordagem baseada em PG, que está sendo proposta, usa um conjunto de alternativas geradas geneticamente. As alternativas podem ser muito diferentes do programa original, permitindo o teste de interação entre defeitos, com a vantagem de ser independente da linguagem e paradigma de implementação utilizados. Neste trabalho são apresentados dois procedimentos de teste baseados em PG para selecionar e avaliar dados de teste e é descrita a ferramenta GPTesT (***G**enetic **P**rogramming based **Tes**ting **T**ool*), implementada para validar a abordagem proposta. Resultados de um experimento com a GPTesT, também, são apresentados.

Abstract

Genetic Programming (GP) has recently been applied to solve problems in diverse areas. It has the goal of inducing programs from fitness cases by using the concepts of Darwin's evolution theory. On the other hand, software testing, that is a fundamental and expensive activity to assurance software quality, has the goal of generating test cases from the program being tested. In this sense, a symmetry between induction of programs based on GP and testing is noticed. This work explores such symmetry and proposes a GP approach for software testing. Fault-based testing criteria generally derive test data using a set of mutant operators to produce alternatives that differ from the program under testing by a simple modification. The GP approach, herein proposed, uses a set of alternatives genetically derived. The alternatives can be very different from the original program allowing the test of interactions between faults. We present two GP-based testing procedures respectively for selection and evaluation of test data sets and we describe GPTesT (***G**enetic **P**rogramming based **Tes**ting **T**ool*), a framework implemented to validate the proposed approach. Results from an experiment with GPTesT are also presented. The approach based on GP can be used as a testing criterion, and is independent on the language or paradigm based to implement the program being tested.

Sumário

1 Introdução *1*

1.1 Contexto **1**

1.2 Motivação **4**

1.3 Objetivos **5**

1.4 Organização **5**

2 Sobre o Teste de Software *7*

2.1 Introdução **7**

2.2 Etapas do Processo de teste **8**

2.3 Técnicas e Critérios de Teste de Software **9**

2.3.1 Técnica Funcional 10

2.3.2 Técnica Estrutural 11

2.3.3 Técnica Baseada em Erros 12

2.3.3.1 Análise de Mutantes 13

2.3.4 Outros Critérios de Teste 15

2.4 Ferramentas de Teste **16**

2.4.1 PokeTool 16

2.4.2 Mothra 17

2.4.3 PROTEUM 17

2.5 Principais Limitações da Atividade de Teste **18**

2.5.1 Correção Coincidente 19

2.5.2 Caminhos não Executáveis 20

2.5.3 Mutantes Equivalentes 20

2.5.4 Caminhos Ausentes 21

2.6 Comparações entre Critérios **23**

3 Programação Genética *25*

3.1 Computação Evolucionária **25**

3.2 Algoritmos Genéticos **26**

3.2.1 Elementos de um Algoritmo Genético 27

3.2.1.1 População 27

3.2.1.2 Indivíduo 27

3.2.1.3 Função de *Fitness* 28

3.2.1.4 Seleção 28

3.2.1.5 Reprodução 28

3.3 Programação Genética **29**

3.3.1 Programação Genética Baseada em Gramática 33

3.4 Ferramenta Chameleon **36**

3.5 Outras Ferramentas de Programação Genética **42**

3.5.1 GPC++ 42

3.5.2 LIL-GP 42

4 Programação Genética e a Atividade de Teste *43*

4.1 Seleção de Casos de Teste **44**

4.1.1 Exemplo Utilizando Chameleon para Seleção de Casos de Teste.. 45

4.2 Avaliação de Casos de Teste **48**

4.2.1 Exemplo Utilizando Chameleon para Avaliação de Casos de Teste 49

4.3 Comparativo **50**

5 Ferramenta GPTesT *53*

5.1 Procedimentos de Teste usando GPTesT **59**

5.1.1 Exemplo de Seleção de Casos de Teste 59

5.1.2 Exemplo de Avaliação de Casos de Testes 66

6 Utilizando GPTesT *69*

6.1 Experimento com a GPTesT 69
6.1.1 Descrição do Experimento 69
6.1.2 Análise dos Resultados 72
6.2 GPTesT X Proteum 74
6.2.1 Descrição do Experimento 74
6.2.2 Análise dos Resultados 76
7 Conclusões e Trabalhos Futuros 78
Referências Bibliográficas 81
Apêndice A.Programas utilizados no Experimento com a GPTesT 87
Apêndice B. Procedimento com resultado de cada passo realizado com os programas no emprego de GPTesT 91
Apêndice C. Resultados da execução dos programas com a GPTesT e a Proteum 124

Lista de Figuras

Figura 2.1: Exemplo de Grafo de Fluxo de Controle. 11

Figura 2.2: Exemplo de Correção Coincidente. 19

Figura 2.3: Código de Programa com Caminho Não Executável. 20

Figura 2.4: Exemplo de Programas Equivalentes. 21

Figura 2.5: Exemplo de Caminho Ausente. 22

Figura 3.1: Estrutura de um Algoritmo Genético 26

Figura 3.2: Exemplo de Cruzamento em Programação Genética 32

Figura 3.3: Exemplo de Gramática. 34

Figura 3.4: Exemplo de um Indivíduo Obtido com Gramática, "x + x * x" 35

Figura 3.5: Diagrama Funcional da Ferramenta Chameleon. 37

Figura 3.6: Configuração Inicial para a Chameleon. 39

Figura 3.7: Exemplos de Possíveis Soluções Encontradas pela Chameleon. 40

Figura 4.1: Procedimento de Seleção de Casos de Teste. 45

Figura 4.2: Código do Programa Fatorial. 45

Figura 4.3: Gramática para o Problema Fatorial. 46

Figura 4.4: Procedimento para Avaliação de Casos de Teste. 49

Figura 5.1:: Diagrama de Casos de Uso - GPTesT 54

Figura 5.2: Diagrama Ferramenta GPTesT. 58

Figura 5.3: Código do Programa MMC 60

Figura 5.4: Configuração Inicial / para Chameleon, Caso MMC. 62

Figura 5.5: Exemplos de Alternativas Geradas. 63

Figura 5.6: Resultados da GPTesT. 65

Figura 5.7: Alternativas Vivas para o MMC. 66

Figura 5.8: Resultado Final do caso MMC. 66

Figura 5.9: Código do Programa Maior de Três Números 67

Figura 5.10: Resultado para o Conjunto de Casos de Teste *T*, no Caso Maior de Três Números. 68

Figura A.1: Código do Programa Fatorial 87

Figura A.2: Código do Programa Maior de Três Números 88

Figura A.3: Código do Programa MMC 89

Figura A.4: Código do Programa MDC 90

Figura B.1: Configuração Inicial *I* para o Programa Fatorial 92

Figura B.2: Resultado Calculado para a Medida de Cobertura 96

Figura B.3: Código do Programa Fatorial Alterado 97

Figura B.4: Resultado da Medida de Cobertura 99

Figura B.5: Configuração Inicial *I* 102

Figura B.6: Medida de Cobertura – Primeiro Resultado para o Maior de Três Números 106

Figura B.7: Medida de Cobertura - Resultado Final para o Maior de Três Números 107

Figura B.8: Configuração inicial *I* para o Programa MMC 111

Figura B.9: Medida de Cobertura para o MMC 114

Figura B.10: Configuração Inicial *I* para o Programa MDC 118

Figura B.11: Medida de Cobertura para o MDC – Primeiro Resultado 121

Figura B.12: Medida de Cobertura para o MDC – Resultado Final 122

Lista de Tabelas

Tabela 2.1: Exemplos de Operadores de Mutação. 14

Tabela 2.2: Exemplos de Casos de Teste 19

Tabela 3.1: Exemplo dos Arquivos "function.cpp" e "function.h" Usados pela Chameleon. 41

Tabela 4.1: Parâmetros – Configuração Inicial *I*. 46

Tabela 5.1: Informações Iniciais para GPTesT 61

Tabela 5.2: Conjunto de Casos de Teste *T* 68

Tabela 6.1: Exemplos de Alternativas Geradas por Chameleon 70

Tabela 6.2: Principais Resultados Obtidos com a GPTesT 71

Tabela 6.3: Resultados Obtidos com a GPTesT e a Proteum 75

Tabela 6.4: Casos de Teste Efetivos 77

Tabela B.1: Casos de teste para o Programa Fatorial 91

Tabela B.2: Exemplo de Alternativas Geradas por Chameleon 93

Tabela B.3: Dados de teste para a GPTesT – Programa Fatorial 94

Tabela B.4: Resultados da Execução do Programa em Teste com os Casos de Teste 95

Tabela B.5:Número de Alternativas Mortas com a Execução dos Casos de Teste 95

Tabela B.6: Alternativa para o Cálculo do Fatorial 96

Tabela B.7: Resultados da Execução do Programa em Teste Alterado 98

Tabela B.8: Número de Alternativas Mortas na Execução....................................98

Tabela B.9: Resultados Programa Fatorial..100

Tabela B.10: Casos de Teste para o Programa Fatorial.....................................100

Tabela B.11: Casos de Teste 101

Tabela B.12: Alternativas Geradas para o Problema Maior de Três Números.....103

Tabela B.13: Dados de Teste para GPTesT – Programa Maior de Três Números...104

Tabela B.14: Resultados Obtidos com a Execução do Programa Maior de Três Números...105

Tabela B.15: Número de Alternativas Mortas – Primeiros Casos de Teste Incluídos...105

Tabela B.16: Casos de Teste Incluídos e Número de Alternativas Mortas..........106

Tabela B.17: Alternativas para Encontrar o Maior de Três Números..................107

Tabela B.18: Resultados Programa Maior de Três Números..............................108

Tabela B.19: Casos de Teste para o Programa Maior de Três Números............109

Tabela B.20: Casos de Teste para o Programa MMC...110

Tabela B.21: Exemplo de Alternativas Geradas para o MMC.............................112

Tabela B.22: Dados de Teste para a GPTesT – Programa MMC........................112

Tabela B.23: Resultados da Execução do Programa em Teste para o MMC......113

Tabela B.24: Alternativas Mortas após sua Execução...114

Tabela B.25: Possível Alternativa para o Cálculo do MMC.................................115

Tabela B.26: Caso de Teste Incluído para o MMC..115

Tabela B.27: Resultados Programa MMC...116

Tabela B.28: Casos de Teste para o Programa MMC...116

Tabela B.29: Casos de Teste para o Programa MDC...117

Tabela B.30: Exemplo de Alternativas Geradas para o MDC....119

Tabela B.31: Dados de teste para a GPTesT – Programa MDC....120

Tabela B.32: Resultados da Execução do Programa em Teste para o Problema MDC....120

Tabela B.33: Número de Alternativas Mortas com a Execução da GPTesT para o Caso MDC....121

Tabela B.34: Alternativas Possíveis para o Cálculo do MDC....122

Tabela B.35: Caso de Teste Incluído para o MDC....122

Tabela B.36: Resultados Programa MDC....123

Tabela B.37: Casos de Teste para o Programa MDC....123

Tabela C.1: Resultados do Programa Fatorial Executado com GPTesT e Proteum....124

Tabela C.2: Casos de Teste para o Programa Fatorial em GPTesT e Proteum..125

Tabela C.3: Resultados do Programa Maior de Três Números Executado com GPTesT e Proteum....125

Tabela C.4: Casos de Teste para o Programa Maior de Três Números em GPTesT e Proteum....126

Tabela C.5: Resultados do Programa MMC Executado com GPTesT e Proteum....127

Tabela C.6: Casos de Teste para o Programa MMC em GPTesT e Proteum....127

Tabela C.7: Resultados do Programa MDC Executado com GPTesT e Proteum....128

Tabela C.8: Casos de Teste para o Programa MDC em GPTesT e Proteum....128

1 Introdução

1.1 Contexto

A Programação Genética é uma área da Computação Evolucionária, inspirada na teoria da evolução das espécies [8]. É uma técnica com o objetivo de gerar automaticamente programas, através da criação e manipulação de software geneticamente, com o emprego de conceitos obtidos da biologia.

A idéia da Programação Genética é, a partir de dados iniciais, gerar programas na linguagem empregada, com o objetivo de encontrar o programa que melhor resolva um problema específico, conforme estabelecido nos dados iniciais [26].

A geração dos novos programas, partindo de uma geração inicial de programas completamente aleatória, é realizada pela aplicação de operadores genéticos que efetuam uma recombinação entre as partes dos programas. Os operadores genéticos podem ser, entre outros, [26]: reprodução; cruzamento; e, mutação.

Além dos operadores genéticos é empregada uma função, denominada função de *fitness* (ou de adaptação) que irá guiar a seleção dos programas que mais se aproximam da solução.

Para melhorar a qualidade de um software em desenvolvimento, a Engenharia de Software, por sua vez, propõe o uso de técnicas, critérios e ferramentas que auxiliam na produção de software. Dentro desse contexto de busca por qualidade, a atividade de teste tornou-se fundamental e tem sido alvo de muitas pesquisas nas últimas décadas.

O valor do processo de teste no desenvolvimento de software está em gerar produtos confiáveis e examinar sua qualidade [38]. Este processo envolve a

execução do programa a ser testado, a geração de um conjunto de casos de teste e o exame dos resultados obtidos associados aos esperados.

Alguns autores mencionam que há uma simetria entre a atividade de teste e a indução de programas [2, 5, 47]. Isto é, a atividade de teste tem por um de seus objetivos gerar casos de teste a partir de um programa *P* em teste, e técnicas de indução de programas, tais como a Programação Genética, têm o objetivo de gerar programas a partir de casos de teste.

A execução de teste em um software é realizada com o objetivo de revelar defeitos existentes [35]. Para que os defeitos sejam encontrados é preciso que o processo de teste seja bem planejado e que o conjunto de casos de teste seja adequado para que o software possa ser considerado confiável, ou seja, com um alto nível de correção.

Para auxiliar na execução dos testes existem três grupos de técnicas de teste.

1. Técnica Funcional: usa a especificação funcional do programa para derivar casos de teste.
2. Técnica Estrutural: deriva casos de teste baseada nos caminhos dados pelo grafo de fluxo de controle do programa.
3. Técnica Baseada em Erro: deriva casos de teste para mostrar a presença ou ausência de defeitos típicos no programa.

As técnicas, geralmente, são associadas a critérios de teste. Estes são predicados que devem ser satisfeitos para que a atividade de teste seja considerada finalizada, ou seja, quando o programa é considerado testado o suficiente. Os critérios auxiliam as etapas de seleção e análise de adequação de um conjunto de casos de teste. Entre os critérios existentes podemos citar a Análise do Valor Limite, que é um critério funcional [38]; critérios estruturais, tais como: Baseados em Fluxo de Controle e Baseados em Fluxo de Dados [29, 32, 40]; e, critérios baseados em erros: Semeadura de Erros [4] e Análise de Mutantes [12]. Esses últimos critérios têm se mostrado mais eficazes em termos do número de erros

revelados, porém mais custosos com relação ao número de casos de teste e execuções do programa [31, 51].

O critério Análise de Mutantes considera que muitos programadores fazem seus programas muito próximos do programa correto, de acordo com a especificação, isto é chamado de "hipótese do programador competente" [12]. Quando se testa um programa *P*, usa-se o programa correto que se tem em mente e se *P* não estiver correto existe um conjunto de alternativas para *P* que pode incluir pelo menos um programa correto.

Baseado nessa idéia, o critério Análise de Mutantes [12] possibilita o exame do programa em teste, através da geração de programas mutantes. Um programa mutante é criado através de uma alteração sintática simples no programa em teste, definida por operadores de mutação. Este teste é realizado com o fornecimento dos casos de teste pelo testador e a execução dos mutantes para a verificação de seus resultados em relação ao programa em teste. Com os resultados obtidos pode-se verificar quais mutantes chegaram a resultados diferentes do programa em teste. Estes mutantes são considerados mortos, ou seja, retirados do conjunto de mutantes em teste na próxima execução. A finalidade é obter uma medida de cobertura muito próxima do valor 1 (um). A medida de cobertura indica a adequação do conjunto de teste e é aprimorada com a inserção de novos casos de teste, por exemplo.

Alguns trabalhos [11, 22] assumem somente a condição de necessidade para descobrir defeitos, isto é, para revelar um defeito, ou matar um mutante, é necessário produzir somente um estado intermediário diferente entre o programa em teste e suas alternativas, após o comando modificado. Isto porque determinar as condições de suficiência, que são condições para produzir um estado final diferente, é uma questão indecidível (o que está relacionado com o termo correção coincidente [5]). Morell [36], entretanto, aponta que assumir o efeito do acoplamento e considerar somente condições de necessidade pode ter como conseqüência ignorar o efeito global dos defeitos ou as interações de

modificações no programa. Outros trabalhos [44] mostram que na maioria das vezes condições de necessidade não são suficientes.

Outra questão relacionada aos critérios de teste é que para a aplicação dos mesmos é importante a utilização de ferramentas automatizadas, que permitam o seu uso em programas simples e complexos. No caso da Análise de Mutantes, por exemplo, existem ferramentas que apóiam sua aplicação, como a Proteum [10], que testa programas em linguagem C e a Mothra [13, 24], que trabalha com programas escritos em Fortran. No entanto, existem limitações para a completa automatização da atividade de teste, como por exemplo, o problema da identificação de programas equivalentes e da geração de casos de teste.

1.2 Motivação

Do contexto descrito acima, têm-se abaixo os pontos que constituem motivações para o presente trabalho:

- A importância da atividade de teste de software como uma das etapas de desenvolvimento de software;
- A demonstração de que a atividade de teste e a programação genética são campos de estudo que podem ser trabalhados juntos, explorando a simetria entre eles;
- Critérios baseados em erros que distinguem *P* de suas alternativas têm se mostrado os mais eficazes, porém custosos e consideram o efeito do acoplamento, que pode impedir o teste entre interação de defeitos;
- Condições de necessidade nem sempre são suficientes [44];
- Uma ferramenta para apoiar a atividade de teste é fundamental, principalmente, para a seleção e avaliação dos dados de teste, visto que a completa automatização do teste não é possível;

1.3 Objetivos

Este trabalho explora a simetria existente entre teste de software e programação genética (PG), e propõe uma abordagem para seleção e avaliação de dados de teste baseada em PG, denominada GPBT (*Genetic **P**rogramming **B**ased **T**esting*).

A abordagem, analogamente ao critério Análise de Mutantes baseado em erros, utiliza um conjunto de alternativas para realizar a seleção e avaliação de dados no teste de um programa *P*. Porém, as alternativas são obtidas através de Programação Genética.

O resultado é que as alternativas geradas podem ser bastante diferentes do programa em teste e permitir que diferentes tipos ou possíveis interações de defeitos em *P* sejam testados. Essa abordagem também é independente de linguagem ou paradigma de implementação.

Para validar a abordagem e permitir que ela seja utilizada na prática, uma ferramenta denominada GPTesT (***G**enetic **P**rogramming based **Tes**ting **T**ool*) foi implementada e é descrita nesse trabalho. Resultados da utilização da GPTesT são apresentados e comparados com resultados obtidos com a ferramenta Proteum, que apóia o critério Análise de Mutantes.

1.4 Organização

Nesse capítulo foi apresentado o contexto associado ao trabalho proposto, a motivação e os objetivos para sua realização.

No Capítulo 2, é introduzida uma visão geral sobre o teste de software, sendo explorado especialmente o critério Análise de Mutantes.

O Capítulo 3 apresenta os principais algoritmos e ferramentas de Programação Genética.

No Capítulo 4 é abordada a Programação Genética e sua relação com a Atividade de teste, descrevendo dois procedimentos de seleção e avaliação de casos de teste.

O Capítulo 5 apresenta a ferramenta GPTesT, contendo os diagramas de casos de uso e de classes.

O Capítulo 6 contém a descrição dos experimentos realizados com a GPTesT e a Proteum, sendo realizada a análise dos resultados.

No Capítulo 7 estão as conclusões, contribuições e trabalhos futuros.

O trabalho possui três apêndices. O apêndice A contém os programas que foram utilizados para a realização do experimento e os apêndices B e C contêm os resultados obtidos, de modo mais detalhado, com a utilização, respectivamente, da GPTesT e da Proteum.

2 Sobre o Teste de Software

2.1 Introdução

O processo de teste de software tem o objetivo de revelar, tantos quantos possíveis, defeitos no programa em teste [35]. O software é testado pela sua execução, observando-se se com um determinado dado de entrada, o dado de saída obtido é o esperado.

Através desse processo pretende-se aumentar a confiabilidade na correção do software testado. Isto quer dizer, que através de testes não é possível provar a correção de um programa, mas pode-se contribuir para que a confiança no software, em relação ao desempenho das funções especificadas e a qualidade do produto, seja acentuada [30].

Myers [35] cita regras que podem ser empregadas como objetivos de teste.

- Testar é um processo de execução de um programa com a intenção de encontrar defeitos.
- Um bom caso de teste[1] é aquele que tem alta probabilidade de encontrar defeitos desconhecidos.
- Um teste bem sucedido é aquele que revela um defeito ainda não descoberto.

---

[1] Um caso de teste é um par ordenado (d, f(d)), onde d pertence a D (Domínio) e f(d) é a saída esperada, ou seja é o dado de entrada, de um programa, associado à sua respectiva saída.

Em se tratando de teste de software, é importante o esclarecimento de alguns termos e conceitos [23], que serão utilizados ao longo desse trabalho.

- Defeito (*fault*) - passo, processo, problema ou definição incorreta no programa em teste.
- Engano (*mistake*) - ação humana que produz resultado incorreto.
- Erro (*error*) - diferença entre o valor obtido e o esperado.
- Falha (*failure*) - produção de uma saída incorreta com relação a especificação.

2.2 Etapas do Processo de teste

O processo de Teste de Software deve ser dividido em etapas: planejamento de teste, projeto de casos de teste, execução dos testes e avaliação dos dados resultantes. As etapas devem ser desenvolvidas durante o processo de desenvolvimento de software.

Com um bom planejamento pode-se descrever passo-a-passo como os testes serão conduzidos, bem como, ter uma previsão dos esforços, tempo e recursos necessários.

Em um processo de teste devem ser considerados testes de baixo nível, para verificação de um pequeno segmento de código (observando se o mesmo está corretamente implementado), e testes de alto nível, para verificar a validade das funções do sistema a partir dos requisitos e necessidades estabelecidas pelo usuário.

As fases que podem contemplar testes de baixo nível e de alto nível são denominadas:

- Teste de Unidade: teste que concentra esforços na menor unidade do projeto de software, ou seja, em cada módulo separadamente;
- Teste de Integração: teste aplicado durante a integração da estrutura do programa, para descobrir erros relacionados às interfaces entre os módulos;

- Teste de Validação: teste realizado após a integração do sistema, visando a identificação de erros de funções e características de desempenho incoerentes com a especificação.
- Teste de Sistema: testa o software com outros elementos do sistema.

Para projetar casos de teste deve-se pensar nos objetivos, sendo que o conjunto de casos de teste deve ter a mais alta probabilidade de encontrar muitos defeitos com uma quantidade mínima de tempo e esforço. Devido ao tempo e esforço, sabe-se que é impraticável empregar o domínio de dados de entrada completo para avaliar os itens funcionais e operacionais de um software em teste.

Sendo assim, na realização de testes é necessário considerar duas questões [29, 40].

- Como os dados de teste devem ser selecionados?
- Como saber se um programa *P* foi suficientemente testado?

Visando a auxiliar na decisão dessas questões, critérios de teste foram propostos para selecionar e avaliar conjuntos de casos de teste, de modo que os escolhidos sejam aqueles que aumentem as chances de revelar defeitos, ou pelo menos, aumentem o nível de confiança na correção do programa [4, 12, 29, 32, 40].

Um critério é um predicado utilizado para conduzir a atividade de teste e avaliar um conjunto de dados de teste *T*, oferecendo uma medida de cobertura e quando satisfeito, a atividade de teste pode ser encerrada [29, 40].

2.3 Técnicas e Critérios de Teste de Software

Existem basicamente três técnicas para o teste de produtos de software.

- Funcional (teste de caixa preta) → na qual deve-se conhecer as funções especificadas, para as quais o produto foi projetado para executar. Os testes

deverão demonstrar que cada função é inteiramente operacional ao mesmo tempo em que procura por defeitos em cada uma delas.

- Estrutural (teste de caixa branca) → Conhecendo a estrutura interna do produto, pode-se verificar com os testes se a operação interna está sendo executada de acordo com a especificação e se todos os componentes estão trabalhando adequadamente.
- Baseada em Erros → na qual os critérios e requisitos de teste vêm do conhecimento dos erros típicos, que geralmente ocorrem durante o processo de desenvolvimento do software.

2.3.1 Técnica Funcional

O Teste de Caixa Preta ou Funcional é utilizado para demonstrar que as funções do software são operacionais, que a entrada é propriamente aceita e a saída corretamente produzida, e, que a integridade da informação externa é mantida. Neste tipo de teste não há preocupação com os detalhes de implementação.

Os tipos de defeitos que podem ser identificados, em teste de caixa preta, são: funções incorretas ou perdidas, erros de interface, erros na estrutura de dados ou no acesso externo à base de dados, erros de performance, erros de terminação ou inicialização. Este teste tende a ser aplicado no último estágio da atividade de teste.

Para realização do teste de caixa preta é necessário identificar as funções que o software deve executar e criar casos de teste para checar estas funções.

Entre os Critérios de Teste Funcional estão: o Particionamento em Classes de Equivalência [38]; e o Grafo de Causa-Efeito [35].

Uma desvantagem na realização do teste funcional está na especificação do problema, que pode ter sido feita de modo descritivo e não formal, o que torna os requisitos de teste um pouco imprecisos, informais e difíceis de automatizar. Uma vantagem é a necessidade de identificar apenas as entradas, a função a ser computada e a saída, o que o torna praticável em todas as fases de teste.

2.3.2 Técnica Estrutural

O Teste de Caixa Branca ou Estrutural examina detalhadamente os procedimentos. Os caminhos lógicos são testados por casos de teste que exercitam conjuntos específicos de condições e/ou laços ou caminhos associados a um grafo de fluxo de controle do programa (Figura 2.1).

O método de testes de caixa branca pode derivar, por exemplo, casos de teste que: garantem que todos os caminhos independentes dentro de um módulo foram acionados pelo menos uma vez; verificam se todas as decisões lógicas, nos lados verdadeiros ou falsos, são executadas; executam todos os loops no seu limite e dentro do seu limite operacional; exercitam as estruturas de dados internas para assegurar sua validade; exercitam a definição e o uso de variáveis.

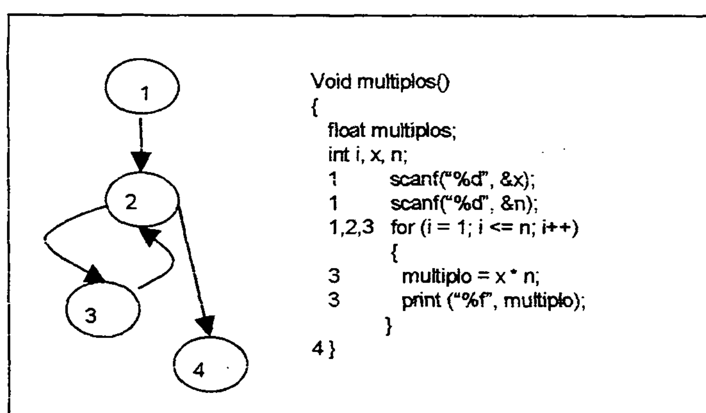


Figura 2.1: Exemplo de Grafo de Fluxo de Controle.

Critérios de Teste Estrutural são, geralmente, classificados em:

- ❖ Critérios Baseados em Fluxo de Controle → consideram características de controle de execução do programa. Os mais conhecidos são: Todos-Nós, no qual a execução do programa deve passar pelo menos uma vez em cada vértice do grafo de fluxo; Todos-Arcos, no qual cada desvio de fluxo de controle deve ser executado pelo menos uma vez; e Todos-Caminhos, no qual todos os caminhos possíveis do programa devem ser executados [40].

- Critérios Baseados em Fluxo de Dados → consideram informações do fluxo de dados, tais como definição de variável e referências a tal definição. Exemplos são: Critérios de Rapps e Weyuker [39, 40], que utiliza o Grafo Def-Uso, que é a adição no grafo de programa de informações sobre o fluxo de dados do programa - definição e uso de variável; Critérios Potenciais-Usos [29]: que testam a partir de um nó i, onde ocorreu uma definição da variável x, caminhos livres de definição com relação à x, sem exigir a ocorrência de um uso explícito.

- Critérios Baseados na Complexidade → consideram informações sobre a complexidade do programa. Um exemplo é o Critério de McCabe [32] ou Complexidade Ciclomática, que é uma métrica de software que dá uma medida quantitativa da complexidade lógica de um programa. Esta medida auxilia na definição do número de caminhos independentes no conjunto básico de um programa e permite a determinação de um número limite para o número de testes que devem ser conduzidos para que sejam executados todos os arcos pelo menos uma vez [38].

2.3.3 Técnica Baseada em Erros

Este tipo de técnica se baseia nos erros mais freqüentes (ênfase aos erros de programação ou projeto) que geralmente ocorrem durante o processo de desenvolvimento de software.

Entre os critérios de teste baseado em erro, pode-se citar, Semeadura de Erros e Análise de Mutantes. No critério de teste Semeadura de Erros, os defeitos são artificialmente introduzidos no programa. O objetivo é que ao se testar o programa obtenha-se uma medida do número de defeitos artificiais pelo número de defeitos naturais revelados. Essa medida pode fornecer uma indicação do número de defeitos ainda existentes no programa [4]. O Critério Análise de Mutantes é de particular interesse neste trabalho e será descrito com mais detalhe.

2.3.3.1 Análise de Mutantes

O Critério Análise de Mutantes foi criado na década de 70, na *Yale University* e *Georgia Institute of Technology* [12], e baseia-se na idéia de que a qualidade de um conjunto de teste está associada à habilidade do conjunto de casos de teste diferenciar o programa *P*, em teste, de um conjunto de programas mutantes, que são gerados a partir de *P* através de uma alteração sintática, e são supostamente incorretos. A diferença é obtida quando o caso de teste produz saídas diferentes para *P* em comparação com os programas mutantes.

Um dos pressupostos da Análise de Mutantes é a hipótese do programador competente, que diz que programadores escrevem seus programas muito próximos do programa correto [12]. Em conformidade com essa hipótese, os defeitos nos programas existem devido a pequenos desvios sintáticos que produzem um resultado incorreto.

Outro pressuposto é o efeito de acoplamento [37], que diz que defeitos complexos estão relacionados a defeitos simples. Portanto, espera-se que conjuntos de casos de teste que revelem defeitos simples também possam revelar defeitos complexos. Assim sendo, na Análise de Mutantes aplica-se uma mutação de cada vez, ou seja, cada mutante difere do programa em teste em apenas uma alteração sintática simples.

Na geração dos mutantes, são aplicados operadores de mutação no programa em teste *P*, isto é, regras que definem as alterações que devem ser aplicadas em *P*. Esses operadores devem induzir mudanças sintáticas simples, baseadas em erros típicos de programadores, ou forçar determinados objetivos de teste como, por exemplo, obrigar a execução de cada arco do grafo de fluxo de controle. Exemplos de operadores de mutação são apresentados na Tabela 2.1.

Tabela 2.1: Exemplos de Operadores de Mutação.

Descrição do Operador de Mutação
Substitui um operador relacional por outro operador relacional.
Substitui operador lógico por operador *bitwise*.
Substitui uma constante por outra constante.
Interrompe a execução do laço após duas execuções.
Troca atribuição aritmética por outra atribuição aritmética.
Substitui o comando *While* por *Do-while*.

Depois os mutantes são executados com um conjunto de casos de teste *T*, com o objetivo de se obter apenas mutantes mortos (mutantes que produzem resultados diferentes do programa *P*) e equivalentes (mutante que computa o mesmo resultado que o programa *P* para qualquer dado de entrada). Dessa forma, o conjunto de casos de teste é adequado ao programa em teste.

Para determinar o nível de confiança da adequação dos casos de teste empregados durante o teste, ou seja, se obter uma medida de cobertura do Critério Análise de Mutantes, é feito um cálculo que determina o escore de mutação, que relaciona o número de mutantes mortos com o número de mutantes gerados. Este escore varia entre 0 e 1, sendo que quanto mais próximo de 1, mais adequado é o conjunto de casos de teste.

O cálculo para o escore de mutação é o seguinte:

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

onde:

DM(P,T) é o total de mutantes mortos pelo conjunto de casos de teste *T*.

M(P) é o total de mutantes gerado a partir do programa *P*.

EM(P) é o total de mutantes equivalentes ao programa *P*.

Um grande número de mutantes vivos implica em um baixo escore de mutação, geralmente atribuído para um conjunto de teste de baixa qualidade. Alguns dos mutantes vivos podem ser equivalentes ao programa testado, e não podem ser mortos. Tais mutantes não contribuem para a suficiência do conjunto de teste e na maioria dos casos devem ser detectados pela pessoa que está realizando o teste.

A análise da cobertura é muito utilizada na condução e avaliação da atividade de teste, e, consiste basicamente em determinar o percentual de elementos requeridos por um dado critério de teste que foram exercitados pelo conjunto de casos de teste utilizado [30].

Com este percentual, pode-se aprimorar o conjunto de casos de teste, através da inserção de novos casos de teste que possam exercitar os elementos não cobertos anteriormente.

Um dos problemas na realização de Análise de Mutantes é o alto custo, pois o número de mutantes gerados pode ser muito grande, o que exige muito tempo de execução.

Na tentativa de reduzir o custo associado à utilização do Critério Análise de Mutantes, foram criados critérios alternativos associados ao original, nos quais procura-se selecionar apenas um subconjunto do total de mutantes gerados, mas preocupando-se em não reduzir a eficácia do critério. Entre esses critérios estão: Mutação Aleatória, no qual seleciona-se aleatoriamente um percentual de mutantes para serem gerados com cada operador de mutação; e, Mutação Restrita, que seleciona operadores de mutação específicos para a geração dos mutantes [30].

2.3.4 Outros Critérios de Teste

Observa-se também outros tipos de critérios de teste. Por exemplo, critérios de geração de seqüências de teste com base em máquinas de estados finitos [7, 15,

19], aplicados em teste de sistemas reativos e orientados a objetos. Os trabalhos citados têm como objetivo distinguir possíveis incorretas máquinas de estado infinito, similarmente ao critério Análise de Mutantes. Bergadano e Gunetti [2] mostram como técnicas de aprendizado baseado em Programação Lógica Indutiva (ILP), também são utilizadas para o teste de programas *Lisp*. Quando a confiabilidade de software é crítica, alguns autores sugerem que diferentes versões do software sejam produzidas e utilizadas no teste funcional do mesmo [3, 25].

2.4 Ferramentas de Teste

2.4.1 PokeTool

Potential Uses Criteria Tool for Program Testing (PokeTool) [6] foi desenvolvida na Faculdade de Engenharia Elétrica da Universidade Estadual de Campinas - UNICAMP. Esta ferramenta de teste apoia a utilização do critério Potenciais-Usos e de outros critérios estruturais de teste baseados em fluxo de dados, como: Todos-Nós e Todos-Arcos.

No princípio foi desenvolvida para o teste de unidade de programas escritos na linguagem C, mas já existem configurações para o teste de programas em Cobol e Fortran. A ferramenta está disponível para ambientes DOS e UNIX.

Para utilizar a ferramenta, o usuário fornece o programa que será testado, o conjunto de casos de teste e seleciona todos ou alguns dos critérios disponíveis, que são: Todos-Potenciais-Usos, Todos-Potenciais-Usos/Du, Todos-Potenciais-Du-Caminhos, Todos-Nós e Todos-Arcos [6]. O resultado obtido pelo usuário é o conjunto de arcos primitivos (são os arcos que uma vez executados garantem a execução de todos os demais arcos do grafo de programa), o Grafo Def do programa em teste, o programa instrumentado para teste, o conjunto de associações necessárias para satisfazer o critério selecionado e o conjunto de

associações ainda não exercitadas. A ferramenta PokeTool também fornece a cobertura dos critérios implementados, mediante o fornecimento de um conjunto de casos de teste.

As ferramentas Asset [17] e Atac [21] apóiam a utilização dos critérios baseados em fluxo de dados de Rapps e Weyuker [40] respectivamente para programas em linguagem Pascal e C.

2.4.2 Mothra

O desenvolvimento da ferramenta denominada Mothra foi iniciado em 1986 por membros do *George Institute of Technology's Software Engineering Research Center* e da *Purdue University* [24]. É um ambiente de teste de software que apóia a utilização do critério Análise de Mutantes. Foi implementada no sistema operacional Ultrix-32 e tem sido portada para ambientes Unix. Os programas testados estão escritos na linguagem Fortran.

Mothra possui um conjunto de 22 operadores de mutação, que geram os mutantes a serem executados para análise do programa a ser testado.

A ferramenta Godzilla [14], que é capaz de auxiliar a etapa de geração de casos de teste próximos da adequação, utilizando um esquema de geração automática de casos de teste baseado em restrições, está acoplada ao Mothra. Isto reduz a interferência humana no ambiente e diminui o custo de aplicação da Análise de Mutantes.

2.4.3 PROTEUM

PROTEUM (Program Testing Using Mutants) [10] é uma ferramenta de testes que apóia a utilização do critério Análise de Mutantes, desenvolvida no ICMC-USP (Instituto de Ciências Matemáticas de São Carlos), que pode ser configurada para trabalhar com muitas linguagens de programação procedimental, tendo a linguagem C como configuração inicial.

A PROTEUM está disponível para os sistemas operacionais SunOS, Solaris e Linux [10].

Esta ferramenta permite ao usuário avaliar a adequação ou gerar um conjunto de casos de teste para um programa, segundo o critério Análise de Mutantes.

A Proteum possui 71 operadores de mutação, classificados em mutação de comandos, de operadores, de variáveis e de constantes.

Com a PROTEUM pode-se definir os casos de teste, executar o programa em teste, selecionar os operadores de mutação, gerar os mutantes, executar os mutantes com os casos de teste, analisar os mutantes vivos e calcular o escore de mutação.

Depois de executar os mutantes, a PROTEUM fornece um resumo estatístico sobre a sessão de teste desenvolvida. Os dados contêm o número total de mutantes, o número de mutantes vivos, o número de mutantes anômalos, o escore de mutação etc.

PROTEUM providencia um relatório sobre a efetividade de cada caso de teste. O testador pode selecionar as informações desejadas no relatório.

A PROTEUM/IM [9] é uma extensão da PROTEUM que implementa um conjunto de operadores de mutação que oferecem a possibilidade de realizar o teste de integração, ou seja, testar a conexão entre as unidades do software. Sua arquitetura e implementação são similares à PROTEUM.

2.5 Principais Limitações da Atividade de Teste

Independentemente da técnica ou critério de teste utilizado, a atividade de teste possui algumas limitações que impossibilitam sua completa automatização. As principais limitações são descritas nessa seção.

2.5.1 Correção Coincidente

Neste caso, o programa contém um defeito que é executado, sendo produzido um estado incorreto, porém, coincidentemente um resultado correto é obtido. Um exemplo de código em que ocorre a correção coincidente é apresentado na Figura 2.2, e na Tabela 2.2 estão casos de teste que demonstram este tipo de limitação.

```
    void calculo(x, y)
    {
    double x, y;
1   if (y < 0)  → if (y <= 0)
2     Printf(x + x + 6 – y * y);
3   else
3     printf(x * x + 3 – y * y);
4   }
```

Figura 2.2: Exemplo de Correção Coincidente.

O programa da Figura acima apresenta um defeito. A condição do comando *if* deveria ser (y <= 0). Para revelar o erro e produzir um resultado intermediário incorreto é necessário que (y < 0) seja diferente de (y <=0), ou se y ≠ 0. Essa é a condição de necessidade. No entanto, muitas vezes a condição de necessidade não é o suficiente para propagar o erro até a saída e produzir uma falha. Para revelar o erro, além de y ≠ 0, tem-se que satisfazer ((x + x + 6 – y * y) ≠ (x * x + 3 – y * y)), que é a condição de suficiência, que como mostrado na Tabela 2.2 revela o defeito. Determinar a condição de suficiência é uma questão indecidível, o que torna a correção coincidente uma limitação na atividade de teste [5].

Tabela 2.2: Exemplos de Casos de Teste.

Caso de Teste	Caminho Esperado	Caminho Percorrido	Saída Esperada	Saída Obtida
(-1, 0)	1 2 4	1 3 4	4	4
(2, 0)	1 2 4	1 3 4	7	10

2.5.2 Caminhos não Executáveis

Um caminho é dito não executável se não existir combinação possível para os valores de entrada, parâmetros e variáveis globais do programa que causam a sua execução [17]. Determinar se um caminho é o não executável é uma questão indecidível [17], ou seja, não existe um algoritmo de propósito geral para determinar se um dado caminho é ou não executável.

Na Figura 2.3 é apresentado um código de programa que possui um caminho não executável. Este caminho é o 1 2 4. Para que esse caminho seja executado é necessário que o laço não seja executado, o que não é possível, dado os valores que a variável *i* assume.

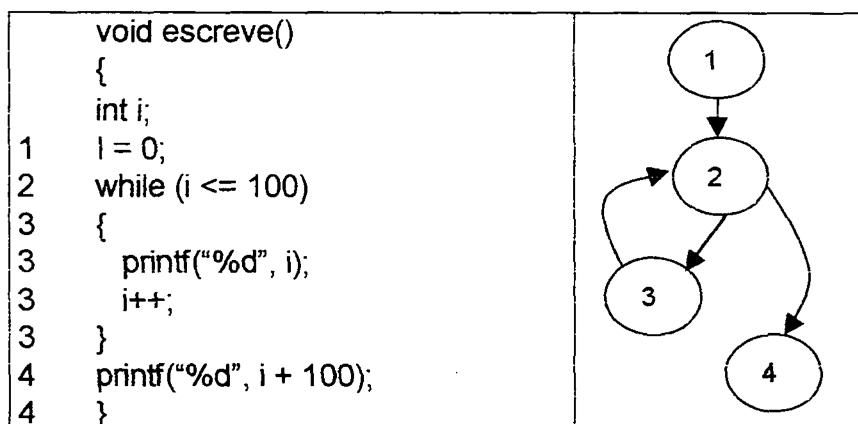


Figura 2.3: Código de Programa com Caminho Não Executável.

2.5.3 Mutantes Equivalentes

É um programa mutante que implementa a mesma função que o programa original *P*, ou seja, não existe um dado de teste capaz de produzir uma diferença de comportamento entre *P* e seu equivalente. Por exemplo, os programas da Figura 2.4 são equivalentes e não existem dados de entrada para diferenciá-los. Em geral, esta é uma questão indecidível e precisa da intervenção da pessoa que está verificando os programas [4].

```
maximo (int a, int b)
{
  int max;
  max = a;
  if (max < b)
    max = b;
  return (max);
}
```

```
maximo (int a, int b)
{
  int max;
  max = a;
  if (max < = b)
    max = b;
  return (max);
}
```

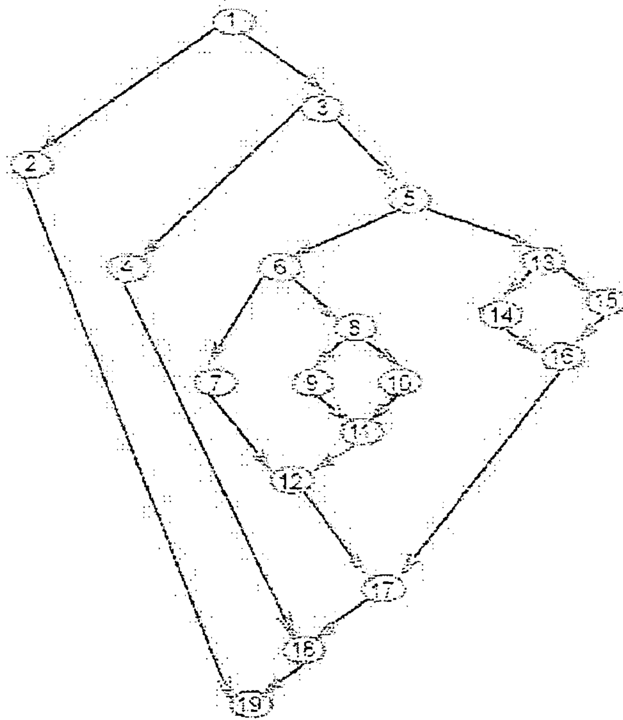
Figura 2.4: Exemplo de Programas Equivalentes.

2.5.4 Caminhos Ausentes

É dito caminho ausente àquele que deveria existir no programa porque corresponde a uma funcionalidade que deveria ser testada, no entanto está ausente. Observe o programa triângulo da Figura 2.4a [41] e o seu gráfico de fluxo de controle (Figura 2.4b). Este programa recebe como entrada 3 números inteiros, a, b, c, que correspondem aos lados de um triângulo. Se os números não representam um triângulo a saída é 0, se eles não estão em ordem decrescente a saída é –1, e se eles formarem um triângulo eqüilátero, isósceles, retângulo, obtusângulo ou acutângulo, as saídas são respectivamente 1, 2, 3, 4, 5. Suponha que o caminho 1 3 4 18 19 está ausente, o teste "ser ou não ser um triângulo" não teria sido implementado. É possível que todos os caminhos do grafo sejam executados e que esse defeito não seja revelado.

```
void triangulo(a,b,c)
int a,b,c;
{
  int class;
  float as,bs,cs;
  if ((a<b) || (b<c))
    class = -1;
  else
  { if (c<=0)
    class = 0;
   else
    { if ((a!=0) && b!=0))
      {
        as = a*a;
        bs = b*b;
        cs = c*c;
        if (as == bs+as)
          class = 3;
        else
          {
          if (as<bs+cs)
            class = 4;
          else
            class = 5;
          }
      }
      else
      { if ((a=b) && (b=c))
          class = 1;
        else
          class = 2;
      }
    }
  }
  printf("%d", class);
}
```

a) código



b) grafo de fluxo de controle

Figura 2.5: Exemplo de Caminho Ausente.

2.6 Comparações entre Critérios

Quando se pretende empregar critérios de teste de modo a obter o melhor resultado com o menor custo, surge uma certa dificuldade para que se saiba qual dos critérios deve ser utilizado. Devido a isto têm sido realizados estudos teóricos e empíricos que procuram, através da comparação entre os critérios, esclarecer esta questão [28, 29, 31, 45, 47, 48, 51].

Estes estudos são norteados pelos fatores básicos [51]:

1. custo: é o esforço necessário na aplicação de um critério, dado pelo número de casos de teste ou métricas dependentes do critério;

2. eficácia: é a capacidade de um critério em detectar um maior número de defeitos em relação a outro;

3. dificuldade de satisfação ou *strength:* é a probabilidade de satisfazer um critério tendo satisfeito outro.

Este último fator está relacionado à relação de inclusão entre critérios. Um critério C_1 inclui um critério C_2, se para todo programa, todo conjunto de dados de teste que satisfaz C_1 também satisfaz C_2. Se nem C_1 inclui C_2 e nem C_2 inclui C_1, então C_1 e C_2 são incomparáveis [39].

Os critérios funcionais, estruturais e baseados em erro são considerados complementares, na literatura, isto porque eles podem revelar defeitos de tipos diferentes [38]. Os critérios baseados em fluxo de dados são mais fortes que critérios baseados em fluxo de controle, isto porque, em geral, os primeiros incluem os segundos [28, 39]. Os critérios estruturais e Análise de Mutantes são teoricamente incomparáveis, somente estudos empíricos podem apontar o relacionamento entre esses critérios [31, 51].

Os estudos empíricos [28, 45, 48] que comparam o critério análise de mutantes com outros critérios, considerando os fatores citados acima, têm demonstrado que

este critério é mais eficaz, isto é, ele proporciona uma maior probabilidade de revelar defeitos. Porém, por outro lado tem um maior custo para aplicação, em termos de casos de teste necessários. Outras desvantagens são o número de execuções solicitadas e a existência de programas equivalentes.

Os critérios estruturais também têm suas desvantagens: aprender os seus conceitos não é muito fácil, e eles, em geral, requerem elementos não executáveis [17, 45]. Isto é similar à limitação dada por programas equivalentes. Outra limitação inerente à atividade de teste e, principalmente, ao critério estrutural é o problema de caminhos ausentes.

3 Programação Genética

Inicialmente, para que seja explicada a Programação Genética, é necessário que sua origem seja abordada.

A idéia principal vem da Seleção Natural apresentada por Darwin [8], na qual verifica-se que sobrevivem os indivíduos que melhor se adaptam ao meio. Esta adaptação vai ocorrendo através de gerações, pela herança de características genéticas. E como conseqüência, é formada uma população de indivíduos selecionados naturalmente, que são os mais bem adaptados ao meio.

Considerando os estudos do ser humano em relação à evolução e sobrevivência natural das espécies, surgiu a Computação Evolucionária.

3.1 Computação Evolucionária

A Computação Evolucionária (CE) [16] é, basicamente, a possibilidade da simulação do processo de evolução natural aplicada à solução de problemas computacionais complexos.

É importante ressaltar que, a idéia da Computação Evolucionária não é reproduzir certos fenômenos que acontecem na natureza, mas utilizar conceitos genéricos que estão presentes nesses fenômenos para a resolução de problemas computacionais.

O princípio da Computação Evolucionária consiste da:

- utilização dos mecanismos de seleção de soluções potenciais e;
- geração de novos indivíduos através da recombinação das características de indivíduos já existentes, de forma semelhante à evolução dos organismos naturais.

A Computação Evolucionária abrange diversos ramos, entre eles estão os Algoritmos Genéticos, Estratégias de Evolução, Programação Evolucionária e a Programação Genética [33], esta última de particular interesse neste trabalho.

3.2 Algoritmos Genéticos

Algoritmos Genéticos foram criados por John Holland [20] para a utilização da evolução natural de Darwin em computadores, com a intenção de realizar experiências de aprendizado de máquina, mas atualmente são muito utilizados para solucionar problemas de otimização.

Algoritmos Genéticos são uma técnica de busca baseada na teoria de evolução natural, consistindo na evolução de uma população de inteiros binários, que são submetidos a transformações unitárias e binárias genéricas e a um processo de seleção.

Um Algoritmo Genético tem a seguinte estrutura (Figura 3.1) [26]:

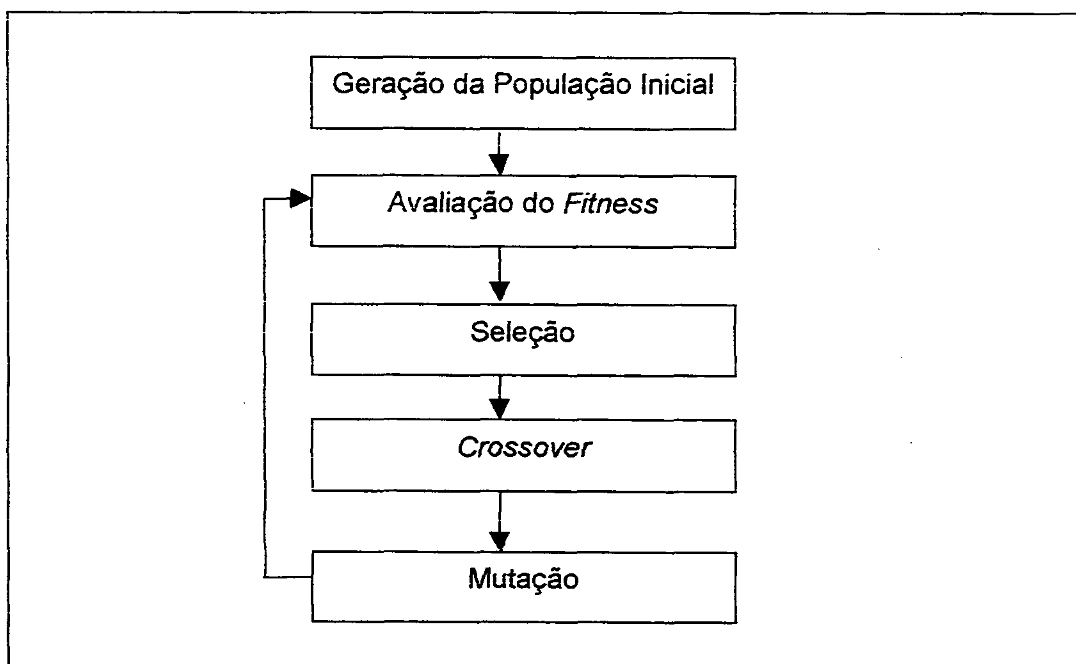


Figura 3.1: Estrutura de um Algoritmo Genético

A execução de um Algoritmo Genético tem por base a geração aleatória de uma população de indivíduos iniciais, considerados possíveis soluções para o problema; a avaliação da população, sendo fornecido para cada indivíduo um valor ou índice que indica sua adaptabilidade a um determinado ambiente (valor de *fitness*); o processo de seleção, pelo qual os indivíduos mais aptos são mantidos e os demais são descartados; o cruzamento (*crossover*) e a mutação, que agem nos indivíduos que sobreviveram a seleção e podem sofrer alterações em suas características fundamentais na geração de novos descendentes. Além do *crossover* e da mutação, os indivíduos selecionados podem ser simplesmente reproduzidos, ou seja, não sofrem nenhuma alteração em suas características.

3.2.1 Elementos de um Algoritmo Genético

3.2.1.1 População

A população é o conjunto de indivíduos da mesma espécie, que vivem numa mesma área e na mesma unidade de tempo, e que está sendo cogitado como solução do problema.

Quando a população é pequena, existem grandes chances de que seja perdida a diversidade necessária na exploração da solução. E, quando a população é grande, pode-se perder eficiência na solução, pela demora em avaliar a função de *fitness.*

3.2.1.2 Indivíduo

O indivíduo é o material genético, composto do conjunto de atributos da solução. Cada atributo é, em geral, representado por uma seqüência de bits, sendo que o individuo é a concatenação das seqüências de bits.

Tradicionalmente, os indivíduos são representados por vetores binários, onde cada elemento de um vetor denota a presença (1) ou ausência (0) de uma determinada característica. Os elementos podem ser combinados formando as características reais do individuo.

3.2.1.3 Função de *Fitness*

A função de *Fitness* (ou função de adaptação) indica a adaptação do indivíduo ou quão boa é a solução a ela associada. Esta função consiste em associar um valor numérico de "adaptação" ao indivíduo, que se supõe ser proporcional à "utilidade" ou "habilidade" do indivíduo representado.

3.2.1.4 Seleção

A seleção é a escolha dos indivíduos que sobrevivem. Ela é realizada para que somente os indivíduos melhores e mais aptos tenham a capacidade de continuarem vivos. Os indivíduos mais adaptados possuem características mais vantajosas para a sobrevivência. Portanto, sempre que ocorre a seleção, os indivíduos mais adaptados deverão ser selecionados em detrimento dos demais.

Entre as estratégias de seleção, pode-se citar as seguintes.

- Torneio → M indivíduos são selecionados aleatoriamente usando uma distribuição uniforme. O melhor dos M é selecionado em relação a *fitness*. O processo é repetido até que seja obtida uma população N.
- Ranking → não paramétrico, sendo que os indivíduos são ordenados de acordo com o seu valor de *fitness*, para que seja feita a seleção.
- Roleta → cada indivíduo da população é representado na roleta proporcionalmente ao seu índice de *fitness*. Assim, os indivíduos com *fitness* alto têm uma porção maior da roleta. A nova população é obtida por N tiragens com reposição considerando que a probabilidade de cada indivíduo é proporcional ao seu valor da função de *fitness*.

3.2.1.5 Reprodução

O processo de reprodução preserva características úteis, podendo introduzir variedades e novidades. Na reprodução podem ser gerados parentes únicos (clonagem + mutação) ou parentes múltiplos (cruzamento + mutação).

O objetivo é a produção de novos cromossomos[2], associando cromossomos selecionados para reprodução, de modo a combinar as boas "qualidades" existentes na população.

O cruzamento (*crossover*) cria novos indivíduos misturando características de dois indivíduos pais. Exemplos de cruzamentos são dados a seguir.

Pai 1: 10101010110101010111 (AB)

Pai 2: 00001001010101110010 (XY)

Cruzamento em um ponto: é selecionado aleatoriamente um ponto de corte no cromossomo. Cada um dos dois descendentes recebe informações genéticas de cada um dos pais. AB e XY geram AY e XB.

10101010110101110010 (AY)

00001001010101010111 (XB)

Cruzamento uniforme: os filhos são formados a partir dos bits dos pais (sorteado)

A mutação é uma operação que inverte aleatoriamente alguma característica do indivíduo, criando novas características que não existiam, de modo que a diversidade da população é mantida. Usando uma representação binária, na mutação, um bit é substituído por seu complemento, ou seja, um "0" é substituído por um "1" e vice-versa.

3.3 Programação Genética

A Programação Genética surgiu a partir da idéia de Algoritmos Genéticos [20, 34], a diferença está nos indivíduos da população, que deixam de ser *strings* de

---

[2] É utilizado o termo cromossomo, em analogia a biologia, para referir-se a indivíduo.

tamanho fixo, passando a ser programas de computador. Portanto, Programação Genética é uma técnica de busca que gera programas de computador para a resolução de um determinado problema [26].

Resumidamente, os passos essenciais na Programação Genética são [27]:

- Geração de uma população inicial de programas, de composição aleatória de funções e terminais do problema;
- execução de cada programa, assinalando seu valor de *fitness*;
- criação de uma nova população de programas de computador pela aplicação das duas operações previstas, reprodução (cópia do programa de computador existente, na nova população) e cruzamento (criação de um novo programa pela combinação genética, escolhida aleatoriamente, de partes dos programas existentes), conforme indicado pela probabilidade baseada no valor de *fitness*;
- o melhor programa de computador que for criado, em qualquer das gerações, é indicado como o resultado da programação genética, podendo ser uma solução ideal ou uma solução aproximada para o problema;
- enquanto a solução não for encontrada ou o número de gerações que serão executadas não for atingido, a execução continua, a partir do segundo passo.

Os indivíduos da população, isto é os programas, são representados, comumente, através de árvore [26], mas existem outras estruturas que incluem genoma linear [1] e árvores de derivação baseadas em gramática [49, 50].

A função de *fitness* direciona a evolução da população e no caso de programação genética, está associada aos casos de treinamento (indicação de valores de entrada e da respectiva saída) utilizados para definir o problema.

Na criação de uma nova população, é feita uma seleção dos indivíduos que serão reproduzidos, da mesma forma que em algoritmos genéticos.

Após a seleção dos indivíduos representados em forma árvore, os novos programas são gerados a partir de duas árvores, que podem ser reproduzidas ou

cruzadas pela remoção de nós de uma delas e inserção dos nós na outra (Figura 3.2). Este processo assegura que o novo programa é uma árvore e também, que é sintaticamente válido. Os elementos que compõem os nós das árvores são funções (comando da linguagem) ou símbolos terminais (parâmetro de um comando). A diferença entre os elementos, é que um terminal tem valor independente, enquanto o valor de uma função é calculado com base nos seus parâmetros. As funções são escolhidas de um conjunto F dado pelo programador. Os terminais são retirados de um conjunto T, que pode conter variáveis, números ou qualquer outro tipo de dado válido.

A mutação, no caso de algoritmos genéticos, é a alteração aleatória de valores de bits e, em programação genética, por exemplo, é a alteração de uma subárvore selecionada aleatoriamente por outra criada, também, aleatoriamente. Este operador ajuda a manter a diversidade na população, isto é importante para que seja evitada a convergência rápida do algoritmo para uma solução simples. No entanto, em programação genética, existem evidências de que a mutação pode ser ignorada [26].

Como a programação genética é um processo evolutivo, em um laço de repetição, é importante que seja estabelecido um critério de parada para sua execução. O critério de parada poderia ser a satisfação de um certo número de dados de treinamento, considerando-se uma margem de erro pré-determinada. Outro critério é o alcance do número de gerações estabelecido para o término da execução. Neste caso, mesmo que a solução aproximada ou ideal não seja atingida, a execução é interrompida.

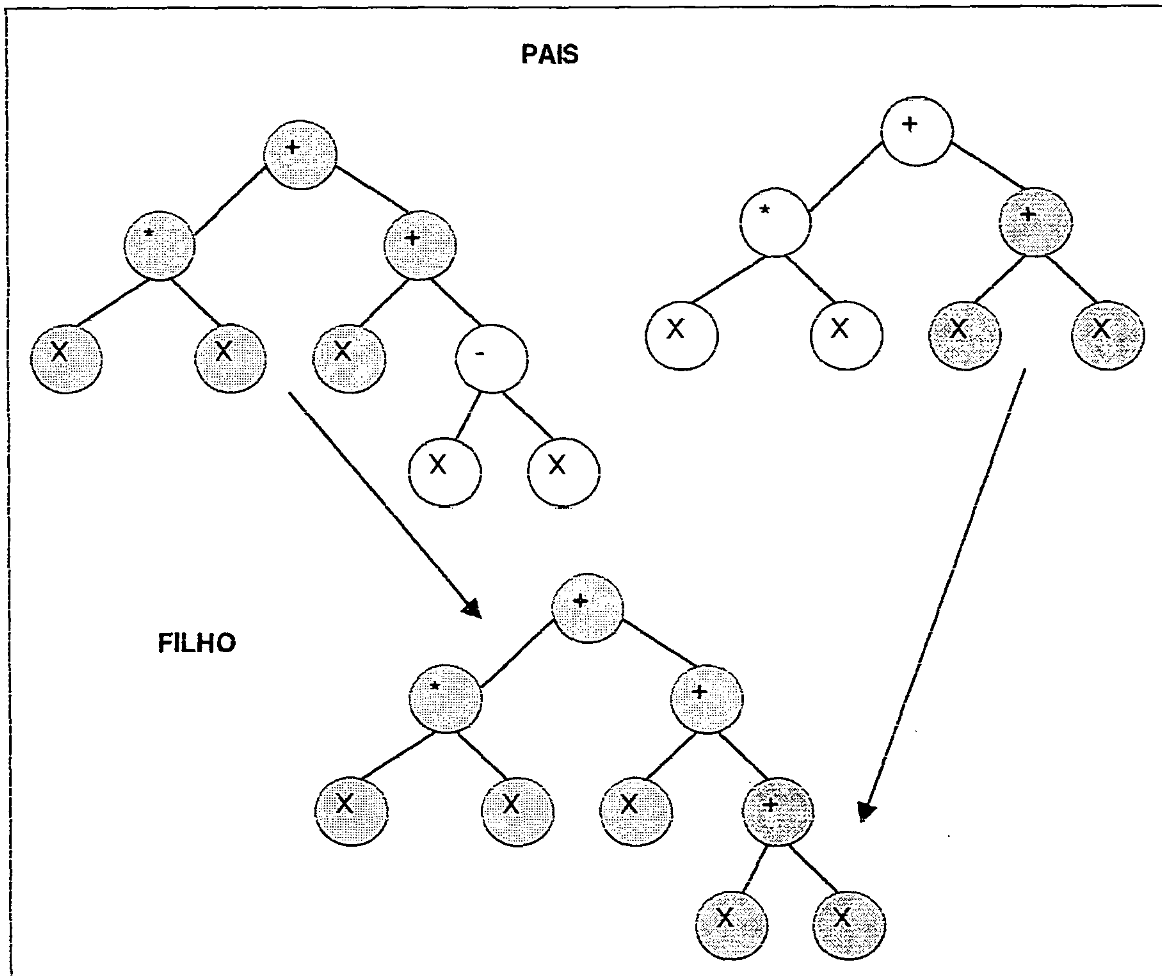


Figura 3.2: Exemplo de Cruzamento em Programação Genética

Para controlar a execução de programação genética, existem alguns parâmetros que devem ser especificados. Os principais são descritos a seguir.

- Tamanho da população: número de indivíduos na população.
- Número máximo de gerações: ponto de parada da execução do programa.
- Probabilidade de cruzamento: valor entre zero e um que indica a realização do cruzamento com os indivíduos selecionados. Exemplo: 90%.
- Probabilidade de reprodução: valor entre zero e um que indica a realização da reprodução com os indivíduos selecionados. Exemplo: 10%.

- Tamanho do torneio: número de programas que constituirão o conjunto de indivíduos a serem selecionados por torneio.
- Profundidade mínima da árvore: número que indica o menor tamanho permitido para os programas gerados.
- Profundidade máxima da árvore: número que indica o maior tamanho permitido para os programas gerados.
- Profundidade máxima de árvore após o cruzamento: número que fixa o tamanho máximo dos programas com a aplicação do cruzamento. Se o tamanho dos programas após o cruzamento for maior que este valor, o cruzamento não é efetivado, e os pais (programas) são reproduzidos.
- Método de inicialização: indicação de como a primeira população será gerada. A escolha é feita entre o método *full*, no qual todos os programas terão o tamanho máximo possível, e o método *grow*, no qual a população inicial pode ter diversos tamanhos, ou ainda, o método *ramped half-and-half*, que envolve o emprego dos dois métodos anteriores para a geração da população inicial [26].
- Margem de erro: valor utilizado para comparar os valores numéricos dos dados de treinamento na definição do ponto de parada da execução da programação genética.

3.3.1 Programação Genética Baseada em Gramática

Programação genética baseada em gramática [49, 50] é uma extensão da Programação Genética original. Utiliza uma gramática para controlar a estrutura dos programas. A gramática serve de base para geração dos indivíduos, isto é, programas que são possíveis soluções para o problema dado.

A seguir pode-se visualizar um exemplo de gramática:

```
<code> ::= <exp>
<exp>  ::= <exp> <op> <exp>
<exp>  ::= <var>
<op>   ::= + | - | * | /
<var>  ::= x
```

Figura 3.3: Exemplo de Gramática.

Programação Genética baseada em gramática utiliza uma árvore de derivação para representar cada indivíduo [49]. Cada nó da árvore de derivação pode ser um terminal, por exemplo, a variável "x", ou um não-terminal, como "<exp>". Um nó terminal possui um valor independente e um nó não terminal depende da avaliação dos seus componentes.

Com base em uma gramática, um indivíduo é criado por meio de uma derivação completa, partindo-se do estado inicial da gramática trabalhada, sendo escolhidos aleatoriamente os nós não-terminais e substituídos por seus valores até que os nós folha sejam nós terminais. As escolhas são feitas aleatoriamente, pois pode existir mais de uma regra possível. Estas operações são chamadas produções. Um exemplo de um indivíduo derivado de gramática é indicado na Figura 3.4.

Similar à Programação Genética tradicional, no processo de evolução, são utilizados operadores genéticos que modificam os indivíduos (programas) através de recombinações feitas diretamente nas árvores de derivação. Os operadores genéticos podem ser:

- *Reprodução* - copia o indivíduo para a nova população;
- Cruzamento - altera a estrutura da árvore de derivação. O cruzamento produz um filho a partir de dois pais – um é dito o primário e o outro o secundário. Uma subárvore é escolhida do primeiro e trocada por uma outra subárvore escolhida do segundo. Contudo, esta escolha deve obedecer às regras de produção da gramática, pois somente podem ser trocados ramos de árvore que tenham sido gerados usando a mesma regra de produção. Por este fato, o cruzamento é dito restrito.

- *Mutação* – também altera a estrutura da árvore de derivação. É feito pela substituição de determinada sub-árvore por outra gerada aleatoriamente. Um nó é selecionado, e a gramática é utilizada para derivar uma nova subárvore para substituir a existente neste nó.

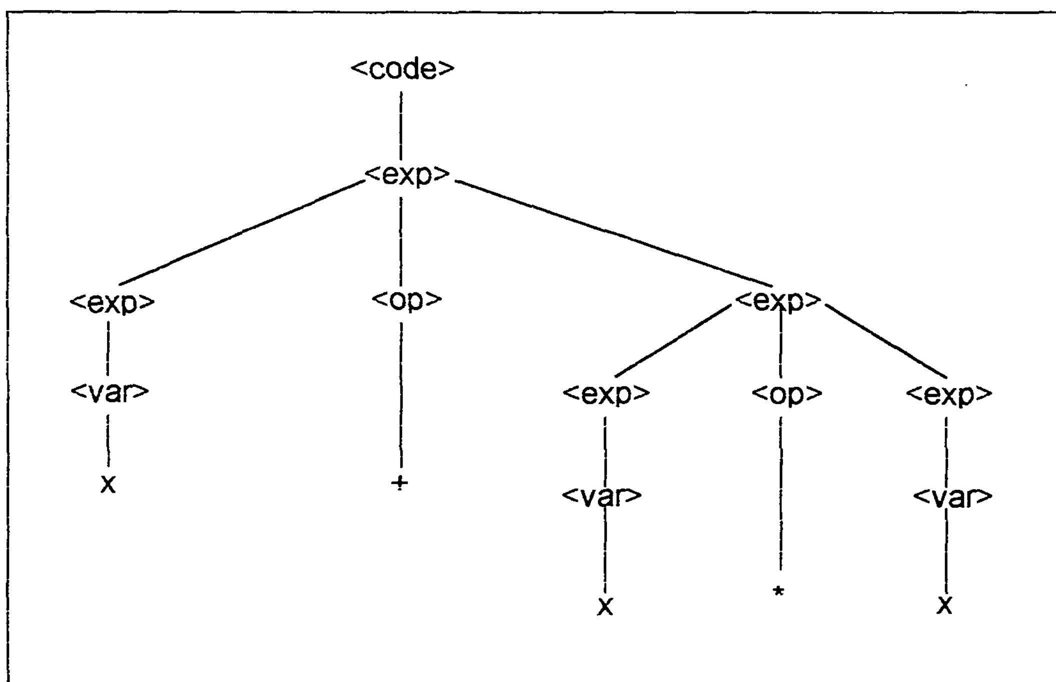


Figura 3.4: Exemplo de um Indivíduo Obtido com Gramática, "x + x * x"

3.4 Ferramenta Chameleon

A ferramenta Chameleon[3] [43] foi desenvolvida na linguagem C++, e projetada para possuir um núcleo genérico e permitir a configuração de diferentes classes de problemas de aplicações da programação genética. O objetivo principal da Chameleon é a disponibilização de uma aplicação para programação genética orientada à gramática, que seja de fácil entendimento e customização, não sendo necessários conhecimentos avançados em C++ para a sua manipulação.

As características mais importantes da Chameleon, são:

- Os programas gerados são avaliados externamente, permitindo flexibilidade quanto à linguagem de programação.
- A definição da gramática, a configuração inicial e os dados de treinamento são fornecidos por um arquivo texto.
- A ferramenta controla a geração de programas anômalos (divisão por zero, laço infinito, e outros), associando aos mesmos um péssimo valor de *fitness*.

A Figura 3.5 apresenta o diagrama funcional da Chameleon.

---

[3] Esta ferramenta foi criada pelo grupo de mestrandos e doutores do Departamento de Informática da Universidade Federal do Paraná, em 2000, durante a Oficina de Inteligência Artificial e Engenharia de Software.

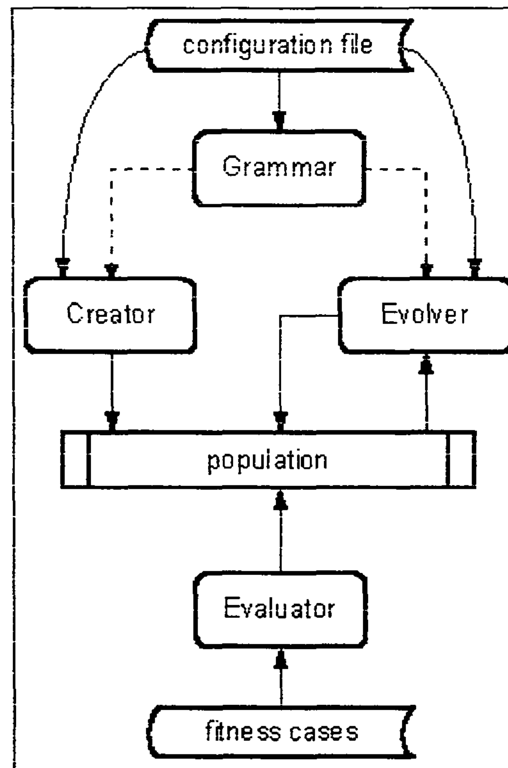


Figura 3.5: Diagrama Funcional da Ferramenta Chameleon.

- *Grammar* → esse componente tem por objetivo principal auxiliar os componentes *Creator* e *Evolver* na geração de programas válidos.
- *Creator* → gera a população inicial com a ajuda da *Grammar*.
- *Evaluator* → executa cada indivíduo da população de acordo com os dados de treinamento empregados para o programa. Também controla a geração de programas anômalos.
- *Evolver* → realiza as operações genéticas, por exemplo o cruzamento, com os indivíduos selecionados para a próxima geração.

A ferramenta Chameleon foi utilizada para realizar alguns experimentos aplicando-se PG para resolver problemas, entre eles, é apresentado o problema do mínimo

múltiplo comum (ou mmc) entre dois números inteiros positivos não nulos, logo a seguir.

Foi definida uma gramática para o problema, que pode ser vista na Figura 3.6, juntamente com os parâmetros necessários para a execução da Chameleon. Além disso, a ferramenta possui dois outros parâmetros que são fornecidos pelo usuário separadamente, que são o número de rodadas e o número de gerações, que foram respectivamente 10 e 100, no caso em questão. Na Figura 3.7, pode-se observar programas gerados pela ferramenta, como possível solução para o problema do mmc.

```
[begin]
[parameters]
population size = 500
tournament size = 10
maximum depth for initial random programs = 15
maximum depth for programs created during the run = 30
crossover rate = 90
mutation rate = 0
elitist = N
threshold = 0.01
[compiler]
cl -nologo -G6 -MT -Fepop.exe
[result-producing branch]
terminal set = {X,Y}
function set = {RESTO,NE,MUL,DIV}
output variable = Z
[result-producing branch productions]
<code>  -> <def> <prog> <result>
<def>   -> float R = 1, A = X, B = Y;
<result> -> Z = DIV(MUL(A,B),<var>);
<prog>  -> if (<expc>) {<prog1>} else {<atr>}
<prog1>  -> do {<bloco>} while (<expc>);
<bloco>  -> <exp>
<bloco>  -> <bloco> <exp>
<exp>   -> <var> = <opm>(<var>,<var>);
<exp>   -> <var> = <var>;
<expc>  -> <opc>(<var>,<cte>)
<atr>  -> <var> = <cte>;
<opm>   -> RESTO
<op>    -> MUL
<op>    -> DIV
<opc>   -> NE
<var>   -> X
<var>   -> Y
<var>   -> R
<cte>   -> 0
[fitness cases]
source -> mmc.dat
[end]
```

Figura 3.6: Configuração Inicial para a Chameleon.

```
float R = 1, A = X, B = Y;if (NE(R,0)) {do {Y =RESTO(X,Y);R
=RESTO(R,R);} while (NE(Y,0));} else {R =0;}Z =
DIV(MUL(A,B),R);

float R = 1, A = X, B = Y;if (NE(R,0)) {do {Y =RESTO(X,Y);R
=RESTO(R,R);} while (NE(Y,0));} else {R =0;}Z =
DIV(MUL(A,B),R);

float R = 1, A = X, B = Y;if (NE(R,0)) {do {R =Y;X =RESTO(X,R);Y
=RESTO(Y,R);} while (NE(X,0));} else {Y =0;}Z =
DIV(MUL(A,B),R);

float R = 1, A = X, B = Y;if (NE(X,0)) {do {R =RESTO(R,R);R =Y;X
=RESTO(X,Y);Y =X;X =R;} while (NE(Y,0));} else {X =0;}Z =
DIV(MUL(A,B),R);
```

Figura 3.7: Exemplos de Possíveis Soluções Encontradas pela Chameleon.

Observando a Figura 3.7, com os programas selecionados pela Chameleon como possíveis soluções para o mmc, percebe-se que os programas resultantes são apenas um fragmento de um código necessário para a execução do problema. Como uma forma de controlar a geração de programas anômalos são definidas operações como, por exemplo, a divisão e o resto, nos arquivos de funções denominados "function.cpp" e "function.h", de modo que não seja possível ocorrer divisão por zero. Exemplificando este fato, o resto é chamado de RESTO(A,B), não sendo utilizado diretamente, resto = A%B. Os arquivos "function.cpp" e "function.h", para o caso mmc, podem ser visualizados na Tabela 3.1. Além disso, outra forma de evitar a geração de programas anômalos é o controle do tempo de execução dos programas, que possui um valor limite, que se ultrapassado o programa é considerado anômalo.

Tabela 3.1: Exemplo dos Arquivos "function.cpp" e "function.h" Usados pela Chameleon.

```
#if !defined(_FUNCTIONS_H)
#define _FUNCTIONS_H
bool NE(float x, float y);
float RESTO(float x, float y);
bool MT(float x, float y);
bool LTE(float x, float y);
float MUL(float x, float y);
float DIV(float x, float y);
#endif
```

arquivo: "function.h"

```
#include "functions.h"

bool NE(float x, float y) {
  return (x!=y);
}

bool LTE(float x, float y) {
  return (x<=y);
}

float RESTO(float x, float y) {
  if (y != 0)
     return (float)((int)x%(int)y);
  else
     return 1;
}

bool MT(float x, float y)
{
  return(x>y);
}

float MUL(float x, float y)
{
  return x*y;
}

float DIV(float x, float y)
{
  if (y != 0)
    return x/y;
  else
    return 1;
}
```

arquivo: "function.cpp"

3.5 Outras Ferramentas de Programação Genética

3.5.1 GPC++

GPC++ é um sistema de Programação Genética de domínio público, desenvolvido em C++ por Fraser [18] e estendido posteriormente [46].

GPC++ requer um bom conhecimento de seu núcleo para ser utilizado na resolução de um problema. Exige um conhecimento bom em C++, especialmente em herança e funções virtuais, devido a manipulação das classes *GPGene*, *GP* e *GPPopulation*.

Além disso, GPC++ só manipula expressões simbólicas de *Lisp* (*S-expressions*).

3.5.2 LIL-GP

LIL-GP [52] é um software gratuito da *Michigan State University* e representa um sistema na linguagem C para o desenvolvimento de aplicações de programação genética.

Existem cinco arquivos que o usuário deve escrever para implementar um problema novo em LIL-GP. Um conjunto de arquivos de usuário é fornecido na distribuição.

Em LIL-GP existem dois tipos de funções, referidas como tipos "*DATA*" e " *EXPR*", porque a ferramenta é baseada em *Lisp*. Em *Lisp o* equivalente a DATA é a implementação de uma função com um "*defun*" e o equivalente a EXPR seria a implementação de uma função com um "*defmacro*".

LIL-GP também manipula só S-expressões de *Lisp*.

4 Programação Genética e a Atividade de Teste

A programação genética, como já foi comentado, é um método capaz de dados parâmetros iniciais e dados de treinamento[4], que indiquem o problema a ser resolvido, criar uma população inicial de programas e evoluí-los até que seja encontrado o melhor programa, ou seja, o que soluciona o problema ou o que mais se aproxima da solução do mesmo.

Na atividade de teste, a meta é gerar casos de teste que permitem revelar o maior número de defeitos possíveis em um programa. Técnicas de teste associadas a um critério de teste devem auxiliar, o testador, em duas tarefas: a seleção e a avaliação de casos de teste.

Dessa forma, programação genética é uma técnica que gera programas a partir de casos de teste e a atividade de teste gera casos de teste a partir de um programa sendo testado. Isto demonstra uma simetria entre testes e programação genética, apontada na literatura [2, 5, 47].

Este capítulo explora essa simetria e apresenta dois procedimentos: um de seleção e outro de avaliação de casos de teste. Esses procedimentos estão baseados em PG e dizem respeito às principais atividades de teste que são, geralmente, guiadas pelos principais critérios de teste descritos no Capítulo 2.

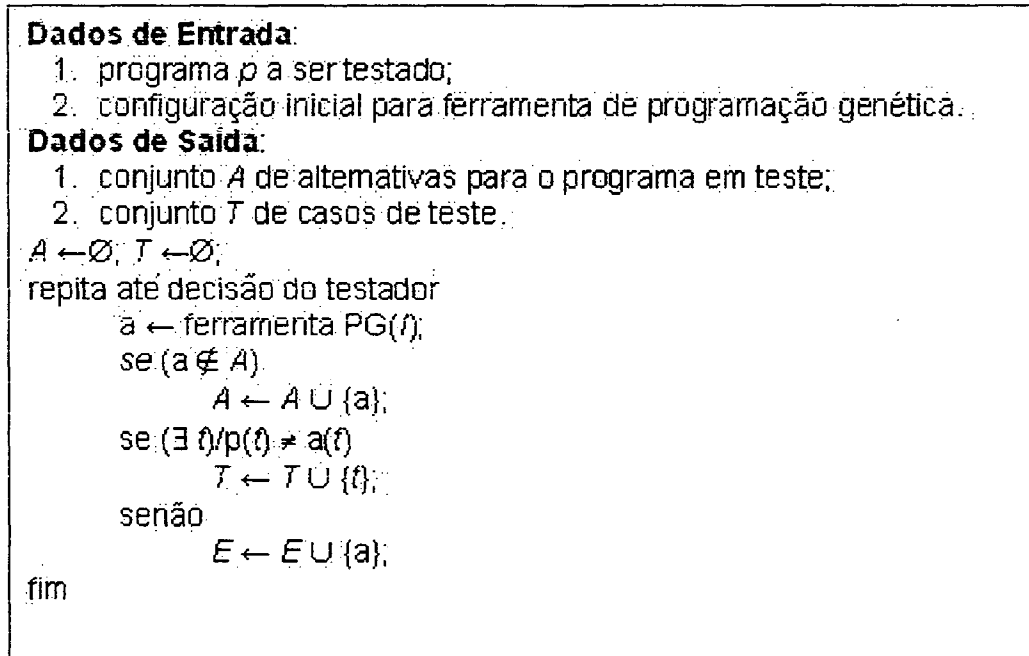
---

[4] Na atividade de teste os dados de treinamento da programação genética são denominados casos de teste. Portanto, no decorrer do trabalho será utilizado o termo casos de teste.

4.1 Seleção de Casos de Teste

O procedimento de seleção de casos de teste utilizando programação genética encontra-se na Figura 4.1. Este procedimento inicia com o conjunto de alternativas de programas *A* e o conjunto de casos de teste *T* vazios. É executado um laço de repetição que gera as alternativas com uma ferramenta de programação genética, que recebe como parâmetro uma configuração inicial *I*, que também contém os dados de treinamento. Primeiramente, verifica-se se a alternativa gerada *a* não pertence a *A*, se não pertencer ela é adicionada a *A*, caso contrário o laço de repetição continua sendo executado, com a geração de outra alternativa. Se *a* foi adicionada a *A*, o testador deve gerar um caso de teste *t* capaz de produzir uma saída diferente para o programa *p* e para *a*. Se isto acontece, *t* é incluído em *T*, caso contrário, se o testador decidir que não existe o caso de teste *t*, *a* é incluído em *E*, conjunto de programas equivalentes.

O conjunto *T* de casos de teste deve ser adicionado à configuração inicial da ferramenta de programação genética, a cada iteração do laço para evitar que a próxima alternativa *a* gerada seja morta por um dado existente em *T*.



```
Dados de Entrada:
  1. programa p a ser testado;
  2. configuração inicial para ferramenta de programação genética.
Dados de Saída:
  1. conjunto A de alternativas para o programa em teste;
  2. conjunto T de casos de teste.
A ←∅; T ←∅;
repita até decisão do testador
        a ← ferramenta PG(I);
        se (a ∉ A)
                A ← A ∪ {a};
        se (∃ t/p(t) ≠ a(t))
                T ← T ∪ {t};
        senão
                E ← E ∪ {a};
fim
```

Figura 4.1: Procedimento de Seleção de Casos de Teste.

4.1.1 Exemplo Utilizando Chameleon para Seleção de Casos de Teste

A seguir é apresentada a execução do procedimento de seleção de casos de teste com o emprego da ferramenta Chameleon, que implementa programação genética.

No exemplo foi utilizado o programa fatorial, visto na Figura 4.2. As alternativas foram geradas por Chameleon. Na Tabela 4.1 é apresentada a configuração inicial *I* com os parâmetros e na Figura 4.3 pode ser observada a gramática *G*.

```
int fatorial(int x)
{ int y,i;
  f = y; i = 1;
  while (i<=x)
  {
    y = y*i;
    i = i+1;
  }
  return(f);
}
```

Figura 4.2: Código do Programa Fatorial.

Tabela 4.1: Parâmetros – Configuração Inicial *I*.

Número de execuções independentes	100
Número máximo de gerações	50
Tamanho da população	1000
Tamanho do torneio	2
Taxa de cruzamento	80 %
Taxa de mutação	10 %
Profundidade mínima da árvore	5
Profundidade máxima da árvore	10
Profundidade máxima permitida	15

```
<code> ::= y = <kte>; while (<expb>) { <stmts> }
<stmts> ::= <stmt>  <stmts>
<stmts> ::= <stmt>
<stmt>  ::= <var> = <var> <opa> <kte>;
<stmt>  ::= <var> = <var> <opa> <var>;
<expb> ::= <var> <opb> <kte>
<var>  ::= x | y
<opb>  ::= < | >
<opa>  ::= * | -
<kte>  ::= 1 | 0
```

Figura 4.3: Gramática para o Problema Fatorial.

Conforme o procedimento descrito anteriormente, *T* está incluído em *I* e inicialmente $T = \varnothing$. Foram seguidos os passos da seguinte forma:

- Primeira iteração do *loop*: Chameleon (G,*I*) é executada, sendo escolhida a alternativa a seguir. Nesse primeiro passo, as alternativas são geradas aleatoriamente.

```
a1(int x)
{ int y;
  y = 0; while ( x > 0) { x = y * y; y = y - 1;}
  return (y);
}
```

O testador gera um dado de teste *t* tal que a1(*t*) ≠ p(*t*); *t* e a1 são incluídos respectivamente em *T* e *A*. Então, *T* = {0} e *A* = {a1}.

- Segunda iteração do *loop*: Chameleon (G,*I*) é executada e a alternativa a2 é escolhida. A alternativa a2 satisfaz *T*.

```
a2(int x)
{ int y;
 y = 1; while ( x > 0) { x = y - 1; y = y * x;}
 return (y);
}
```

O testador gera um dado de teste *t* tal que a2(*t*) ≠ p(*t*); *t* e a2 são incluídos respectivamente em *T* e *A*. Então, *T* = {0,1} e *A* = {a1,a2}.

- Terceira iteração do *loop*: Chameleon (G,*I*) é executada e a alternativa a3 é escolhida.

```
a3(int x)
{ int y;
 y = 1; while ( x < 0) { x = x - x; y = x - y;}
 return (y);
}
```

O testador gera um dado de teste *t* tal que a3(*t*) ≠ p(*t*); *t* e a3 são incluídos respectivamente em *T* e *A*. Então, *T* = {0,1,2} e *A* = {a1,a2,a3}.

- Quarta iteração do *loop*: Chameleon (G,*I*) é executada e o resultado é o seguinte.

```
a4(int x)
{ int y;
 y = 1; while ( x > 1) { y = x * 1; x = x - 1;}
 return (y);
}
```

O testador gera um dado de teste *t* tal que a4(*t*) ≠ p(*t*); *t* e a4 são incluídos respectivamente em *T* e *A*. Então, *T* = {0,1,2,3} e *A* = {a1,a2,a3,a4}.

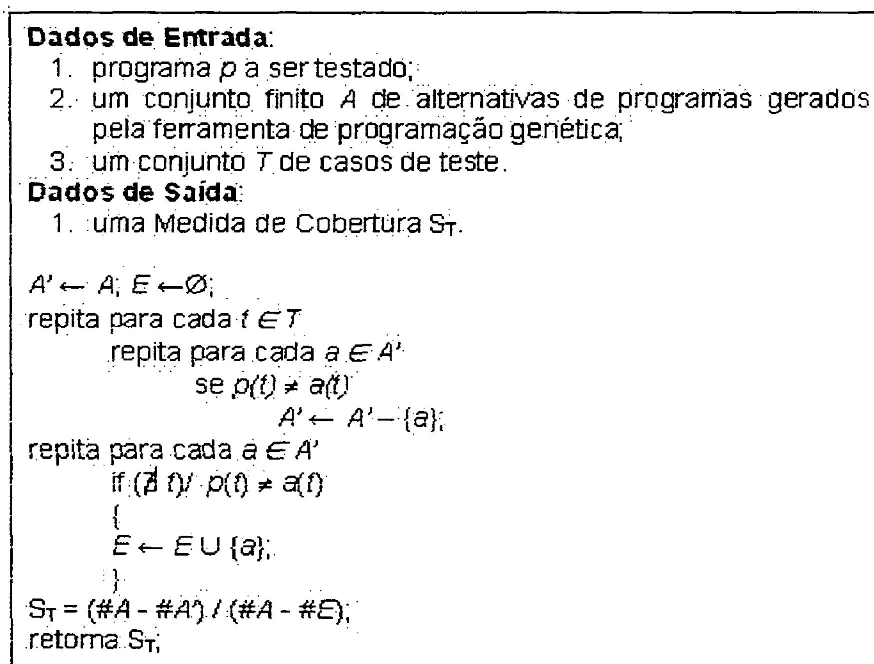
Após essas iterações, o *loop* e o procedimento são encerrados pelo testador que é quem decide parar. A solução é dada por *T* e *A*. Um critério para determinar o ponto de parada do algoritmo é verificar as alternativas geradas pela ferramenta, se após um determinado número de iterações, a ferramenta só produzir alternativas equivalentes, significa que a solução pode ter sido encontrada. Por exemplo, a alternativa a5, a seguir, é equivalente a *p*.

```
a5(int x)
{ int y;
 y = 1; while ( x > 1) { y = x * y; x = x - 1;}
 return (y);
}
```

4.2 Avaliação de Casos de Teste

Um procedimento de avaliação de casos de teste é apresentado na Figura 4.2:

- O programa *p* e as alternativas geradas por uma ferramenta de programação genética são executados com todos os casos de teste previstos pelo testador no conjunto *T*, que está sendo avaliado. Se *p* e *a* produzem saídas diferentes para determinado *t*, *a* é considerado morto e excluído.
- Como um segundo passo o testador deve identificar os programas de *A* equivalentes a *p*. Em *A'* estão somente as alternativas vivas e que são utilizadas para essa análise.
- A medida de cobertura S_T para *T* é calculada similarmente ao cálculo feito no critério Análise de Mutantes.



Dados de Entrada:
1. programa *p* a ser testado;
2. um conjunto finito *A* de alternativas de programas gerados pela ferramenta de programação genética;
3. um conjunto *T* de casos de teste.

Dados de Saída:
1. uma Medida de Cobertura S_T.

```
A' ← A; E ← ∅;
repita para cada t ∈ T
      repita para cada a ∈ A'
            se p(t) ≠ a(t)
                  A' ← A' − {a};
repita para cada a ∈ A'
      if (∄ t)/ p(t) ≠ a(t)
      {
      E ← E ∪ {a};
      }
S_T = (#A - #A') / (#A - #E);
retorna S_T;
```

Figura 4.4: Procedimento para Avaliação de Casos de Teste.

4.2.1 Exemplo Utilizando Chameleon para Avaliação de Casos de Teste

Para a avaliação de casos de teste utilizando Chameleon, são usados o programa fatorial da Figura 4.2 e os mesmos parâmetros que foram utilizados para a Chameleon na seleção de casos de teste.

A proposta baseada em programação genética pode ser usada como um critério para avaliar a qualidade de um conjunto de casos de teste. Por exemplo, considerando dois conjuntos de casos de teste, *T1* ={0,1,2} e *T2* = {0,1,2,3}, a questão seria: Qual é o melhor conjunto?

Para responder esta questão, pode-se usar o conjunto de alternativas geradas por Chameleon, *A* ={a1,a2,a3,a4,a5}, para calcular a medida *S*, e escolher a maior medida. Com a execução do procedimento para os dois conjuntos de casos de teste e o testador determinando alternativas equivalentes, o resultado é o que segue:

T1: Inicialmente $A' = A$ e $E = \varnothing$.

No fim, $E = \{a5\}$ e $S_{T1} = 0,75$.

T2: Inicialmente $A' = A$ e $E = \varnothing$.

No fim, $E = \{a5\}$ e $S_{T2} = 1$.

Dessa forma, o conjunto *T2* é adequado e melhor que *T1*, em relação a *A*, já que S_{T2} é maior que S_{T1}.

4.3 Comparativo

Com relação a aplicação e dificuldade de automação, temos que determinar alternativas equivalentes; esse é um problema similar a determinar caminhos/elementos não executáveis no teste estrutural.

GPBT é supostamente uma abordagem mais cara, pois, assim como o critério análise de mutantes requer um número grande de execuções do programa para a avaliação dos testes e geração das alternativas.

A abordagem de teste baseada em programação genética assemelha-se a critérios baseados em erro, principalmente ao critério análise de mutantes, por isso, pode-se esperar que ela apresente as mesmas vantagens e desvantagens desse critério, com relação aos demais existentes.

Devido a isso, os próximos parágrafos são dedicados à comparação entre Análise de Mutantes e GPBT.

O efeito de acoplamento não é pressuposto pelo GPBT, pois as alternativas geradas por uma ferramenta de programação genética podem ser bem diferentes do programa *P* em teste. Na análise de mutantes, os mutantes são gerados através de operadores de mutação que permitem somente uma alteração sintática simples no programa *P* em teste. O GPBT permite testar interação entre defeitos, pois é possível combinar dois ou mais mutantes como uma única alternativa. Isto pode reduzir o número de mutantes e os esforços gastos pelo usuário. O

procedimento de seleção de casos de teste, apresentado anteriormente, também pode contribuir para este fato, pois alternativas que seriam mortas pelo conjunto *T* corrente não são geradas. Em programação genética, pode ser gerado o mesmo programa mais de uma vez, desde que a escolha da alternativa é aleatória ou baseada na função de *fitness*. Mas na ferramenta que apóia a abordagem de GPBT pode ser implementado um mecanismo que descarte estas alternativas automaticamente, reduzindo o seu número. Além disso, a ferramenta de programação genética que gera as alternativas pode ter implementado um mecanismo que descarte programas anômalos, evitando a geração de alternativas que possam morrer pelos casos de teste existentes e, também, reduzindo o número de alternativas que devem ser analisadas pelo testador. A ferramenta Chameleon faz isto.

GPBT pode utilizar qualquer ferramenta que providencia programação genética, sendo assim, se, por exemplo, a ferramenta for baseada em gramática, as alternativas geradas dependerão diretamente da gramática e conseqüentemente da linguagem de implementação, isto quer dizer, que o testador é quem vai decidir o quanto a hipótese do programador competente irá influenciar na geração das alternativas. Isto não é uma desvantagem, pois os efeitos de um caminho ausente são reduzidos já que o código do programa em teste não é usado para derivar as alternativas. O que é uma vantagem também durante a fase de manutenção.

De forma diferente da análise de mutantes, GPBT é independente da linguagem ou paradigma usado no programa em teste. Observe que um conjunto de operadores de mutação é específico para uma linguagem. Aqui, os procedimentos foram exemplificados, utilizando a ferramenta Chameleon. No entanto, qualquer outra ferramenta de PG poderia ter sido utilizada, tais como as descritas no Capítulo 3, que evoluem programas em Lisp, um paradigma funcional. Para aplicação da abordagem de operador de mutação é necessária a definição dos mesmos para esses tipos de programas.

Com respeito ao procedimento de avaliação de casos de teste, há um importante ponto a ser analisado. A medida é fortemente baseada no conjunto de alternativas

A, que é uma entrada para o procedimento. Medidas diferentes são geradas se conjuntos de alternativas diferentes são considerados. Obviamente que a medida de mutação também depende dos operadores de mutação usados, mas a dependência é diferente. Com duas execuções da ferramenta de programação genética não é garantido que serão geradas as mesmas alternativas que foram geradas anteriormente, mesmo que a configuração inicial para a ferramenta seja a mesma. Isto não é uma desvantagem, pois pode ser um outro fator usado para comparar conjuntos de casos de teste.

No próximo capítulo é apresentada uma ferramenta implementada para apoiar a abordagem de teste baseada em programação genética descrita nesse capítulo e permitir a realização de experimentos.

5 Ferramenta GPTesT

Nesse capítulo é descrita a ferramenta GPTesT (*Genetic Programming based Testing Tool*) que foi implementada para apoiar e validar a abordagem de teste baseado em programação genética, descrita no capítulo anterior.

A ferramenta GPTesT utiliza a ferramenta Chameleon[5], que é responsável pela geração de alternativas que serão utilizadas para o teste de um programa. GPTesT foi desenvolvida em C++, com o Microsoft Visual C++ 5.0, e auxilia no teste de unidade de programas escritos em linguagem C.

Esta ferramenta associada à ferramenta Chameleon apóia a execução da abordagem descrita. Isto quer dizer, que a ferramenta Chameleon é a responsável pela geração dos programas alternativos, por programação genética orientada à gramática. E a ferramenta GPTesT realiza os testes utilizando as alternativas geradas comparando-as com o programa em teste, executando-os, para posterior análise dos resultados, similarmente a ferramenta Proteum [10].

A GPTest é executada tendo como informações iniciais o nome da sessão de teste, o nome do arquivo que contém o código do programa a ser testado, a configuração inicial para a Chameleon e as alternativas geradas.

A configuração inicial para a Chameleon possui parâmetros necessários para a execução da Programação Genética, inclusive a gramática e o nome do arquivo com o conjunto de casos de teste correspondentes ao problema. Alguns dos parâmetros necessários para a execução da Chameleon, que constam na configuração inicial *I*, são: taxa de cruzamento e de mutação, número de rodadas,

---

[5] Descrita em 3.4.

número de gerações, tamanho da população, a descrição da gramática e o nome do arquivo com que contém o conjunto de casos de teste *T*.

Na Figura 5.1, pode ser verificado o diagrama de casos de uso para a GPTesT. E na seqüência uma breve descrição.

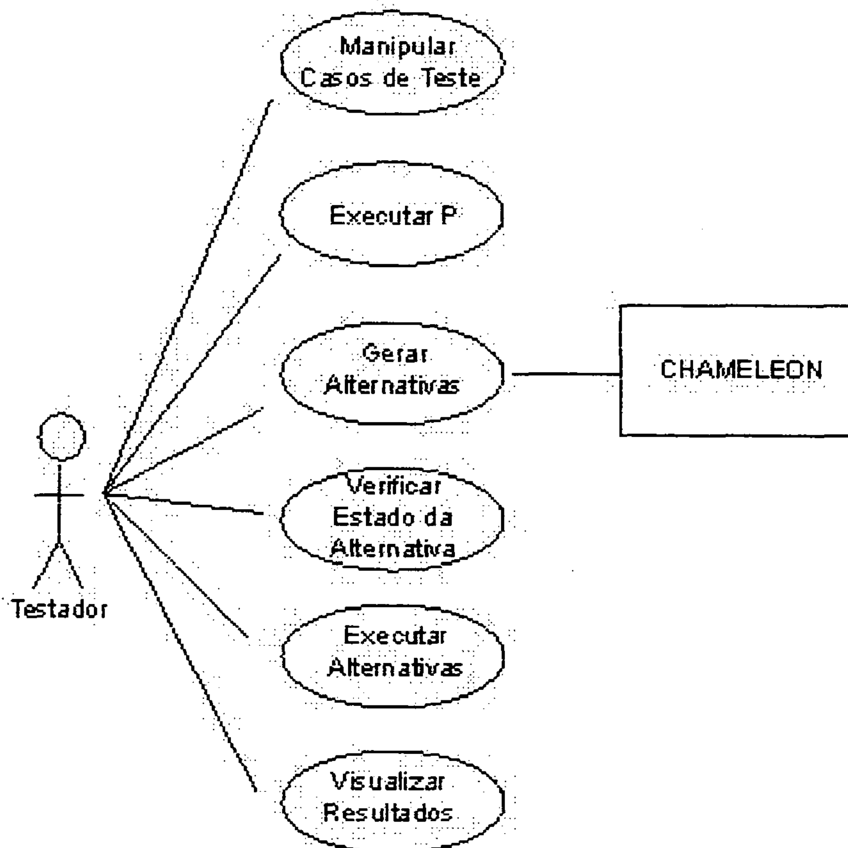


Figura 5.1:: Diagrama de Casos de Uso - GPTesT

Manipular Casos de Teste → através desse caso de uso o usuário adiciona, elimina ou desabilita um caso de teste. Quando é realizada a execução do programa em teste, são visualizados, na tela, os resultados obtidos com os casos de teste correspondentes. Dessa forma, o testador poderá observar se os resultados são os esperados, se não forem, o testador já poderá verificar o programa em teste, que deve conter algum defeito, e corrigí-lo antes de continuar o procedimento de teste. Os casos de teste são salvos em arquivos e associados

à sessão de teste através do arquivo que contém os elementos essenciais da sessão. A opção "Mostrar Sessão" permite ao testador visualizar os casos de teste que já foram inseridos na sessão.

Executar P → executa o programa em teste com os casos de teste não executados até o presente momento. Os resultados são armazenados para que possam ser comparados com os resultados que serão obtidos com a execução das alternativas geradas. Em arquivo, também, pode-se verificar os resultados armazenados da execução do programa em teste com os casos de teste.

Gerar Alternativas → as alternativas de programas para resolver o problema especificado são geradas com o uso da ferramenta Chameleon, que precisa da configuração *I*. A GPTesT receberá da Chameleon, como alternativas geradas os melhores programas de cada geração de uma rodada, que estão armazenados em arquivos. Destes programas são descartados os anômalos e os equivalentes que podem ser sintaticamente determinados.

Verificar Estado da Alternativa → após a execução das alternativas geradas, com um caso de teste, é realizada a comparação dos resultados das alternativas com o resultado do programa em teste, sendo marcadas as alternativas de acordo com o seu estado atual, para que a execução possa continuar com o próximo caso de teste. Estes estados podem ser:

- Anômalo: a alternativa possui um comportamento anômalo, como um laço infinito, uma divisão por zero, etc.
- Morto: a alternativa produziu uma saída diferente para o caso de teste, em comparação com a saída do programa em teste.
- Vivo: a alternativa tem produzido a mesma saída obtida com o programa em teste para os casos de teste executados.

Outro estado possível para uma alternativa é ser equivalente. Para marcar a alternativa como equivalente, o testador deverá após ter executado as alternativas com o conjunto de casos de teste, verificar se as alternativas que permanecem

vivas são equivalentes. Isto pode ser feito pela análise das alternativas, que ainda estão vivas, armazenadas em um arquivo.

Executar Alternativas → executa todas as alternativas, conforme seu estado[6], com os casos de teste que ainda não foram executados.

Visualizar Resultados → o testador pode visualizar informações sobre a sessão de teste e a Medida de Cobertura. A visualização de informações ocorre na abertura de uma sessão e na escolha da opção "Mostrar Sessão".

- Abrir Sessão: ao abrir uma sessão o testador pode ver um relatório com informações como: o nome da sessão; o nome arquivo com o programa em teste; o nome do arquivo com as alternativas geradas (ou programas gerados); os casos de teste (nome do arquivo, valor e estado); os resultados já obtidos na manipulação anterior da sessão (medida de cobertura, número total de alternativas geradas, número total de alternativas anômalas, número total de alternativas vivas e número total de alternativas equivalentes); informações das alternativas geradas (número, estado e código do programa).
- Mostrar Sessão: é visualizado um relatório com informações como: o nome da sessão; o nome do arquivo com o programa em teste; o nome do arquivo com as alternativas geradas (ou programas gerados); os casos de teste (nome do arquivo, valor e estado); informações das alternativas geradas (número, estado e código do programa).
- Medida de Cobertura: esta é uma opção que permite ao testador verificar os resultados calculados, com base na execução do procedimento de teste realizado com a GPTesT. O cálculo da Medida de Cobertura é feito por GPTesT empregando a fórmula:

---

[6] É importante lembrar que somente as alternativas que continuam vivas são executadas com o próximo caso de teste.

$$S_m(P,T) = \frac{A_d(P,T)}{A(P) - A_e(P)}$$

onde:

P é o programa em teste;

T é o conjunto de casos de teste;

$S_m(P, T)$ é a medida de cobertura;

$A_d(P,T)$ é o total de alternativas mortas pelo conjunto de casos de teste *T*;

A(P) é o total de alternativas geradas;

$A_e(P)$ é o total de alternativas equivalentes ao programa *P*.

A Medida de Cobertura indicará a adequação dos casos de teste. A meta é criar um conjunto de teste para que a cobertura se aproxime ao máximo do valor 1.

A Figura 5.2, compreende o diagrama das principais classes da ferramenta GPTesT, e logo a seguir, é apresentada uma descrição sucinta das mesmas.

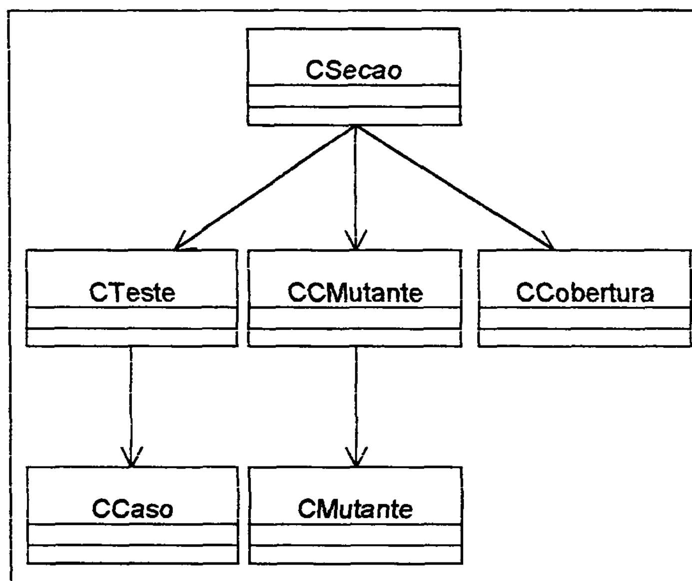


Figura 5.2: Diagrama Ferramenta GPTesT.

- CSecao: a classe CSecao é a classe que contém métodos para verificar a existência e leitura de arquivos; salvar a sessão de teste, exibir as informações e resultados obtidos com a execução do procedimento de teste; compilar e executar o programa em teste, armazenando os resultados; verificar os arquivos com as alternativas geradas pela Chameleon, criando um arquivo que contém somente as alternativas válidas e parâmetros que serão utilizados para a execução dessas alternativas, como: o comando de compilação, as variáveis de entrada e a variável de saída. Esses parâmetros são retirados do arquivo com a configuração inicial *I*, que é utilizado pela Chameleon.
- CTeste: a classe CTeste contém objetos de CCaso e alguns métodos que permitem a manipulação dos casos de teste. Esses métodos são relacionados à leitura e captura dos casos de testes e à alteração de seu estado (habilitado/desabilitado).

- CCaso: cada objeto de CCaso representa um caso de teste a ser testado, ou seja, em CCaso está armazenada a estrutura dos casos de teste, que consta de: número do caso, estado e valor da entrada.
- CCMutante: a classe CCMutante contém objetos de CMutante e alguns métodos que manipulam as alternativas geradas pela Chameleon. Os métodos estão associados a leitura e captura das alternativas geradas, preparação do arquivo com as alternativas para a compilação e execução, compilação e execução das alternativas com os casos de teste ainda não executados, alteração do estado das alternativas (viva, morta, anômala, equivalente) e armazenamento de informações sobre as alternativas que podem ser necessárias posteriormente.
- CMutante: cada objeto CMutante representa uma alternativa gerada pela Chameleon, isto é, em CMutante estão armazenados cada programa gerado que pode ser solução do problema, sendo que cada um contém um número e seu estado. O programa armazenado é somente a parte essencial, o núcleo, do código de um programa propriamente dito. Exemplos podem ser vistos no capítulo que trata do Experimento e na próxima seção.
- CCobertura: a classe CCobertura possui métodos que estão relacionados com a medida de cobertura, como por exemplo, o cálculo deste valor.

5.1 Procedimentos de Teste usando GPTesT

5.1.1 Exemplo de Seleção de Casos de Teste

Para ilustrar o procedimento de Seleção de Casos de Teste foi utilizado o programa Mínimo Múltiplo Comum (MMC), cujo código está na Figura 5.3.

```
/*calcula o mmc entre a e b
# include <stdio.h>
# include <stdlib.h>

main(int argc, char * argv[])
{
  int a, b, resto, A, B; /* a e b contêm os valores iniciais */
if (argc != 3)
 {
        printf("Erro na quantidade de parâmetros. Uso: mmc
<num1> <num2> \n");
        exit(1);
 }
 a = atoi(argv[1]);
 b = atoi(argv[2]);
 A = a;
 B = b;
 if ((a > 0) && (b > 0)){
        do
        {
                resto = A % B;
                A = B;
                B = resto;
        }
        while (resto != 0);
    printf("%d", (a*b)/A);
 }
 else
 {
  printf("0\n");
 }
}
```

Figura 5.3: Código do Programa MMC.

Em se tratando do procedimento de seleção de caso de teste, o testador emprega a GPTesT para guiá-lo nesta atividade. Os passos a serem seguidos pelo testador são:

1. **Inicialização da GPTest**: informações iniciais necessárias para a execução da GPTesT (Tabela 5.1). Destas informações, somente o nome da sessão e o nome do programa em teste *P* deverão ser fornecidos pelo testador para o GPTesT, ao criar uma nova sessão de teste, as outras informações devem estar armazenadas no diretório de trabalho.

Tabela 5.1: Informações Iniciais para GPTesT

Nome da Sessão
Nome do arquivo com o programa em teste – *P*
Arquivo com a configuração inicial *I* da Chameleon[7]
Arquivos com as Alternativas geradas[8]

[7] Arquivo que contém a configuração inicial *I* da Chameleon para o experimento MMC pode ser visto na Figura 5.4.

[8] Arquivos que contêm os programas (alternativas) gerados pela Chameleon, que serão usados no segundo passo do procedimento que está sendo apresentado.

```
[begin]
[parameters]
population size = 500
tournament size = 7
maximum depth for initial random programs = 30
maximum depth for programs created during the run = 60
crossover rate = 90
mutation rate = 0
elitist = N
threshold = 0.01
[compiler]
cl -nologo -G6 -MT -Fepop.exe
[result-producing branch]
terminal set = {X,Y}
function set = {RESTO,NE,MUL,DIV}
output variable = Z
[result-producing branch productions]
<code>  -> <def> <prog> <result>
<def>   -> float R = 1, A = X, B = Y;
<result> -> Z = <op>(<op>(A,B),<var>);
<prog>  -> if (<expc>) {<prog1>} else {<atr>}
<prog1> -> do {<bloco>} while (<expc>);
<bloco> -> <exp>
<bloco> -> <bloco> <exp>
<exp>   -> <var> = <opm>(<var>,<var>);
<exp>   -> <var> = <var>;
<expc>  -> <opc>(<var>,<cte>)
<atr>   -> <var> = <cte>;
<opm>   -> RESTO
<op>    -> MUL
<op>    -> DIV
<opc>   -> NE
<var>   -> X
<var>   -> Y
<var>   -> R
<cte>   -> 0
[fitness cases]
source -> mmc.dat
[end]
```

Figura 5.4: Configuração Inicial I para Chameleon, Caso MMC.

2. **Seleção de Alternativas**: neste caso, foram geradas por Chameleon 526 alternativas de programas, baseando-se na gramática e nos casos de teste dados. A GPTesT, primeiramente, realiza a seleção de alternativas válidas, retirando as equivalentes que podem ser sintaticamente determinadas. Portanto, um conjunto de 44 alternativas continuou sendo considerado. Exemplos das alternativas que permaneceram no teste podem ser vistos na Figura 5.5. As alternativas, como já foi mencionado neste capítulo, não são um código completo de um programa.

```
float R = 1, A = X, B = Y;
if (NE(X,0)) {
  do {
    R =RESTO(R,R);
    R =Y;
    X =RESTO(X,Y);
    Y =X;X =R;
  } while (NE(Y,0));
}
else {
  X =0;
}
Z = DIV(MUL(A,B),R);
```

a)

```
float R = 1, A = X, B = Y;
if (NE(Y,0)) {
  do {
    Y =RESTO(X,Y);
    X =Y;
    R =RESTO(X,R);
  } while (NE(Y,0));
}
else {
  Y =0;
}
Z = DIV(MUL(A,B),R);
```

b)

```
float R = 1, A = X,  B = Y;
if (NE(Y,0)) {
 do {
  R =Y;
  Y =RESTO(X,Y);
  X =R;
 } while (NE(Y,0));
} else {
  X =0;
}
Z = DIV(MUL(A,B),R);
```

c)

```
float R = 1, A = X, B = Y;
if (NE(R,0)) {
  do {
    Y =RESTO(X,Y);
    R =RESTO(Y,R);
  } while (NE(Y,0));
}
else {
  X =0;
}
Z = DIV(MUL(A,B),R);
```

d)

Figura 5.5: Exemplos de Alternativas Geradas.

3. **Geração de casos de teste**: o testador é responsável por selecionar os casos de teste para matar as alternativas. Por exemplo, o caso de teste que mata a alternativa 5.5b é (x = 2 e y = 4), pois os resultados da alternativa e do programa em teste são diferentes.

4. **Execução dos Programas**: as alternativas são preparadas para execução, ou seja, seu código é complementado para que possa ser compilado e executado. Isto é realizado através das informações obtidas com o arquivo de configuração inicial *I* da Chameleon, como o comando de compilação que será utilizado. GPTesT executa *P* e as alternativas com os casos de teste dados pelo testador, gerando resultados como os mostrados na Figura 5.6. A Figura 5.6a mostra um relatório, disponível em arquivo, com o total de alternativas mortas na execução de cada caso de teste. Na Figura 5.6b, vê-se o resultado obtido com o cálculo da medida de cobertura e os valores: número total de alternativas geradas, de alternativas anômalas, de alternativas vivas, de alternativas equivalentes e de casos de testes empregados.

Caso de Teste: 0 - 2 Número Total de Alternativas Mortas: 23 Caso de Teste: 1 - 0 Número Total de Alternativas Mortas: 0 Caso de Teste: 6 - 3 Número Total de Alternativas Mortas: 16 Caso de Teste: 2 - 4 Número Total de Alternativas Mortas: 2 Caso de Teste: 3 - 7 Número Total de Alternativas Mortas: 0 Caso de Teste: 0 -0 Número Total de Alternativas Mortas: 0 Tempo de execução: 00:00:44 h
a)
Total de Alternativas: 44 Total de Alternativas Anômalas: 0 Total de Alternativas Vivas: 3 Total de Alternativas Equivalentes: 0 Total de Casos de Teste: 6 MEDIDA DE COBERTURA: 0.931818
b)

Figura 5.6: Resultados da GPTesT.

5. **Adição de novos casos de testes**: o testador pode continuar a geração de casos de teste, repetindo os passos 3 e 4 até que todas as alternativas não equivalentes estejam mortas ou até que a medida de cobertura esperada seja satisfeita. O testador também deve analisar as alternativas que permanecem vivas, verificando se existem alternativas equivalentes. Isto é realizado manualmente, através dos arquivos que contêm todas alternativas vivas após a execução de determinado caso de teste. Neste caso, o arquivo que seria analisado, contém o conteúdo visualizado na Figura 5.7.

O resultado final obtido para o programa em teste – MMC, as alternativas geradas pela Chameleon, o número total de casos de teste empregados e a análise das alternativas vivas, pode ser visto na Figura 5.8.

O procedimento é finalizado quando o testador obter a meta desejada, ou seja, quando ele considerar que a medida de cobertura obtida foi à esperada e que o conjunto de casos de teste *T* é bom o suficiente.

```
Programas Gerados Vivos
Numero: 15
float R = 1, A = X, B = Y;if (NE(X,0)) {do {R =RESTO(R,R);R =Y;X =RESTO(X,Y);Y =X;X =R;} while (NE(Y,0));} else {X =0;}Z = DIV(MUL(A,B),R);

Numero: 34
float R = 1, A = X, B = Y;if (NE(R,0)) {do {R =X;X =RESTO(Y,X);Y =R;R =RESTO(X,R);} while (NE(R,0));} else {X =0;}Z = DIV(MUL(A,B),Y);

Numero: 36
float R = 1, A = X, B = Y;if (NE(Y,0)) {do {R =Y;Y =RESTO(X,Y);X =R;} while (NE(Y,0));} else {X =0;}Z = DIV(MUL(A,B),R);
```

Figura 5.7: Alternativas Vivas para o MMC.

```
Total de Alternativas: 44
Total de Alternativas Anômalas: 0
Total de Alternativas Vivas: 0
Total de Alternativas Equivalentes: 0
Total de Casos de Teste: 7
MEDIDA DE COBERTURA: 1
```

Figura 5.8: Resultado Final do caso MMC.

5.1.2 Exemplo de Avaliação de Casos de Testes

No caso da avaliação de um conjunto de casos de teste *T*, o testador deverá ter este conjunto *T*, o programa a ser testado *P* e as alternativas geradas pela Chameleon para o problema definido, de modo que a intenção é avaliar o quanto o conjunto *T* é adequado, no caso em questão.

O exemplo será ilustrado com o programa *P*, que retoma o Maior de Três Números, Figura 5.9, e com os casos de teste apresentados na Tabela 5.2.

```
/* Calcula o maior número entre 3 números de entrada */

# include <stdio.h>
# include <stdlib.h>

main(int argc, char * argv[])
{
  int a, b, c;
 if (argc != 4)
 {
   printf("Erro na quantidade de parâmetros. Uso: maior <num1>
<num2> <num3>\n");
   exit(1);
 }
 a = atoi(argv[1]);
 b = atoi(argv[2]);
 c = atoi(argv[3]);
 if ( (a>=b)&&(a>=c) )
   printf("%d\n", a);
 else if ( (b>=c)&&(b>=a) )
   printf("%d\n", b);
 else if ( (c>=a)&&(c>=b) )
   printf("%d\n", c);
}
```

Figura 5.9: Código do Programa Maior de Três Números.

Tabela 5.2: Conjunto de Casos de Teste *T*

Número	a	b	c
1)	0	1	2
2)	1	2	0
3)	1	0	2
4)	4	5	6
5)	5	6	4
6)	6	4	5

Os passos que devem ser seguidos pelo testador no procedimento de avaliação de um conjunto *T*, usando a GPTesT, a princípio são os mesmos do procedimento de seleção, veja a seguir:

1. Inicialização da GPTesT.
2. Seleção de alternativas.
3. Adição de todos os casos de teste do conjunto *T*.
4. Execução de *P* e das alternativas com os casos de testes do conjunto *T*.
5. Determinação de alternativas equivalentes.
6. Análise da medida de cobertura obtida com o conjunto *T*.

Com a execução de P e das alternativas geradas com o conjunto de casos de teste *T*, foi obtido o resultado apresentado na Figura 5.10.

Total de Alternativas: 127
Total de Alternativas Anômalas: 0
Total de Alternativas Vivas: 8
Total de Alternativas Equivalentes: 7
Total de Casos de Teste: 6

MEDIDA DE COBERTURA: 0.99166

Figura 5.10: Resultado para o Conjunto de Casos de Teste *T*, no Caso Maior de Três Números.

O procedimento de avaliação também pode ser utilizado para comparar dois conjuntos de casos de teste, assim como exemplificado na Seção 4.2.1.

6 Utilizando GPTesT

Neste capítulo é descrito um experimento realizado com a ferramenta GPTesT. Esse experimento foi realizado com o objetivo de validar a ferramenta e a abordagem de teste baseada em programação genética apresentada neste trabalho. Os resultados obtidos também puderam ser comparados com o critério Análise de Mutantes implementado pela ferramenta Proteum.

6.1 Experimento com a GPTesT

Primeiramente, é apresentado o procedimento realizado, com os passos seguidos e parâmetros usados; os resultados obtidos; e por fim, a análise destes resultados.

Foram utilizados 4 programas: o fatorial de um número, o maior de três números, o mínimo múltiplo comum de dois números e o máximo divisor comum de dois números. Os códigos fontes utilizados estão no Apêndice A.

6.1.1 Descrição do Experimento

Nesta seção é descrito o experimento realizado com a GPTesT. No apêndice B, pode-se verificar, de forma mais detalhada, os parâmetros usados e resultados de cada um dos programas: a configuração inicial da Chameleon, exemplos de alternativas geradas, casos de teste utilizados e cobertura obtida com cada um deles.

- **Primeiro passo**: executar a ferramenta Chameleon para gerar as alternativas que podem ser solução para o problema em questão.

  Para exemplificar, são mostrados alguns resultados encontrados pela Chameleon em problemas diferentes, na Tabela 6.1.

Tabela 6.1: Exemplos de Alternativas Geradas por Chameleon.

Alternativas Geradas
Programa Fatorial
Y =1;while(MT(Y,0)) {Y =SUB(X,Y);X =MUL(Y,Y);Y =MUL(Y,Y);Y =MUL(Y,Y);Y =MUL(Y,Y);Y =MUL(X,Y); }
Y =1;while(MT(X,0)) {Y =MUL(X,Y);X =SUB(X,1); }
Programa Maior de Três Números
if(MT(C,B)) {if(MT(A,C)) {if(LT(B,C)) {if(LT(B,A)) {if(LT(B,A)) {Z =A;}}}} else {Z =C;}} else {if(LT(B,A)) {Z =A;} else {Z =B;}}
if(LT(A,B)) {if(LT(C,B)) {if(MT(C,C)) {if(LT(B,A)) {if(MT(A,C)) {Z =C;}} else {Z =C;}} else {Z =B;}} else {Z =C;}} else {if(LT(A,C)) {if(LT(C,C)) {if(MT(C,C)) {if(MT(B,A)) {if(LT(A,A)) {Z =B;}} else {if(MT(B,A)) {Z =A;}}}} else {Z =B;}} else {Z =C;}} else {Z =A;}}

- **Segundo passo**: gerar os casos de teste (entradas) e submetê-los à GPTesT para que sejam utilizados na execução do programa em teste e das alternativas, com a finalidade de comparar os resultados, identificando alternativas vivas, mortas, anômalas ou equivalentes.

- **Terceiro passo**: executar o programa em teste e as alternativas geradas com a GPTesT. Ao final algumas alternativas são mortas e uma cobertura inicial é obtida.

- **Quarto passo**: verificar a medida de cobertura e identificar alternativas equivalentes.

➢ **Quinto passo**: adicionar casos de teste, se necessário, até que todas as altemativas não equivalentes sejam mortas.

Os principais resultados obtidos com a execução dos programas na GPTesT estão sumarizados na Tabela 6.2, na qual são apresentadas o número de altemativas geradas pela Chameleon, sendo enfatizado o número de altemativas ativas, que são as utilizadas na execução da GPTesT; o número de alternativas anômalas, se ainda existirem, pois a Chameleon já descartou os anômalos que foram detectados por ela; o número de programas equivalentes, que foram identificados pelo testador; o número total de casos de teste e o número de casos de teste efetivos, ou seja, aqueles que mataram alternativas; e por fim, o tempo total de execução das altemativas. A medida de cobertura obtida em todas as execuções foi 1 (um).

Tabela 6.2: Principais Resultados Obtidos com a Gptest

Programa	Nº Alternativas Geradas	Nº Alternativas Ativas	Nº de Anômalos	Nº de Equivalentes	Nº de Casos de Teste	Nº de Casos de Teste Efetivos	Tempo de Execução Total
Fatorial	814	402	0	0	9	5	00:02:47h
Maior de Três Números	170	115	0	7	18	6	00:03:04h
MMC	525	44	0	0	7	4	00:00:54h
MDC	464	152	0	0	9	7	00:01:38h
Totais	1973	713	0	7	43	22	00:08:23h

6.1.2 Análise dos Resultados

Em relação às alternativas geradas por Chameleon, é importante observar que em dependência da gramática estabelecida e do conjunto de casos de teste (entrada e saída) são gerados programas alternativos mais ou menos próximos de um programa correto para a solução de um dado problema, inclusive considerando-se a hipótese do programador competente.

Para que isso seja sempre possível, é necessário que ao utilizar a GPTesT, o testador, em um primeiro momento, defina uma gramática que realmente seja capaz de tomar possível a resolução do problema que está sendo trabalhado. E, além disso, que o conjunto de casos de teste usado para a geração de alternativas para o programa, contenha possibilidades que possam indicar qual seria a solução do problema.

Isto é dito, porque quanto mais distante da realidade do problema estiver a gramática e o conjunto de casos de teste criados para a Chameleon, mais provável será a possibilidade das alternativas geradas serem mortas no primeiro caso de teste que estiver sendo empregado na execução da GPTesT, o que poderia não levar o testador a detecção de defeitos no código original para a resolução do problema e a não consideração da hipótese do programador competente.

Quanto aos problemas apresentados neste trabalho, pode-se perceber que as gramáticas e casos de teste utilizados com a Chameleon foram gerados na tentativa de expressar os problemas. E sendo assim, em nenhum dos casos observou-se que logo no início da sessão de teste, todas as alternativas foram consideradas inválidas ou mortas.

Em relação ao número de casos de teste usados em cada um dos problemas selecionados para a realização do experimento com GPTesT, pode-se observar que a média geral foi de 10,75. Mas se for considerado o número de casos de

teste que fizeram diferença, ou seja, alteraram o número de alternativas vivas, a média geral é reduzida para 5,5. Portanto, a quantidade de casos de teste necessária foi pequena para que o resultado esperado da medida de cobertura fosse atingido.

O tempo de execução, depende do programa e da quantidade de alternativas geradas ativas, o que se pôde perceber na execução das alternativas pela GPTesT é que o tempo foi baixo.

Outro fator importante, é que o número de alternativas geradas em cada um dos casos não é muito grande, e considerando somente as alternativas válidas (que não são anômalas e nem idênticas), este número se reduz ainda mais. Nenhum programa anômalo foi gerado, sendo que os mesmos foram descartados pela Chameleon. Assim como a abordagem de operadores pode-se optar, se o testador quiser, por selecionar um número maior de alternativas, segundo a configuração utilizada para a Chameleon.

Quanto às alternativas equivalentes, observa-se que somente no problema maior de três números são identificadas alternativas que realmente solucionam o problema e são consideradas equivalentes ao programa em teste. O importante é observar que estas alternativas podem ser bem distintas do programa em teste e, no entanto, também são soluções. Já nos outros casos existiram alternativas que permaneceram vivas por mais tempo, mas acabaram morrendo por um caso de teste com características que não foram observadas para a geração das alternativas.

No caso do cálculo do fatorial, foi possível detectar um erro no programa original devido ao resultado obtido com ele e com as alternativas geradas com determinado caso de teste. Demonstrando que com o emprego da GPTesT pode-se descobrir erros ainda não encontrados no código de um programa (maiores detalhes podem ser vistos no Apêndice B).

Em se tratando dos problemas mínimo múltiplo comum e máximo divisor comum, pode-se observar que as alternativas que permanecem vivas acabam sendo consideradas mortas por casos de teste que têm a mesma característica, ou seja,

números inteiros negativos. Esta é uma característica que não foi observada na geração das alternativas, mas que se fez necessária durante o teste.

6.2 GPTesT X Proteum

O objetivo desse experimento foi realizar uma comparação com o critério Análise de Mutantes, implementado pela ferramenta Proteum, e a abordagem baseada em programação genética. Para isso foram utilizados os programas do experimento anterior e os resultados obtidos com a ferramenta GPTesT.

6.2.1 Descrição do Experimento

O experimento realizado com a Proteum executou os seguintes passos:

- **Primeiro passo**: gerar o conjunto de casos de teste.

  Com o programa a ser testado já definido, o primeiro passo foi a geração do conjunto de casos de teste que seria submetido à Proteum pelo testador, sendo que procurou-se gerar casos de teste independentemente dos utilizados no experimento anterior.

- **Segundo passo**: gerar os programas mutantes com a Proteum.

  Os mutantes foram gerados com a utilização de todos os operadores disponíveis nessa ferramenta.

- **Terceiro passo**: executar o programa em teste e os programas mutantes com os casos de teste submetidos.

➢ **Quarto passo**: verificar o resultado obtido na medida de cobertura e identificar mutantes equivalentes.

➢ **Quinto passo**: adicionar novos casos de teste, se necessário.

➢ **Sexto passo**: comparar com resultados obtidos com a GPTesT em termos de custos e eficácia.

Os dados de teste gerados com a ferramenta Proteum estão detalhados no Apêndice C. Na Tabela 6.3 estão sumarizados os resultados obtidos com a Proteum. Note que a Tabela 6.3 é formada pela Tabela 6.2 com a adição de uma linha para cada programa. Nesta linha são apresentados os resultados da Proteum. Além disso, é retirada da Tabela a seguir, a coluna que trazia o tempo de execução das alternativas, o que será explicado posteriormente.

Tabela 6.3: Resultados Obtidos com a Gptest e a Proteum.

Programa	Ferramenta	Nº Alternativas Geradas	Nº Alternativas Ativas	Nº de Anômalos	Nº de Equivalentes	Nº de Casos de Teste	Nº de Casos de Teste Efetivos
Fatorial	GPTesT	814	402	0	0	9	5
	Proteum	263	263	0	30	9	5
Maior de Três Números	GPTesT	170	115	0	7	18	6
	Proteum	527	527	0	85	35	26
MMC	GPTesT	525	44	0	0	7	4
	Proteum	554	553	1	39	17	14
MDC	GPTesT	464	152	0	0	9	7
	Proteum	397	396	1	36	15	12
Totais	GPTesT	1973	713	0	7	43	22
	Proteum	1741	1739	2	190	76	57

6.2.2 Análise dos Resultados

Por meio dos resultados obtidos com os experimentos, pode-se fazer algumas considerações. Para fazer as considerações, os programas gerados pela Proteum e para a GPTesT são chamados de alternativas.

- Número de Alternativas Ativas: o número de alternativas geradas foi maior em dois casos com a GPTesT (fatorial – 814 e mdc – 464), se for considerado o número de alternativas ativas, apenas para o programa fatorial o número foi maior para a GPTesT. No total a GPTesT gerou a metade das alternativas geradas pela Proteum.
- Geração das Alternativas: gerar as alternativas por gramática difere de gerá-las por operadores de mutação. Isto ocorre porque no primeiro caso as alternativas geradas podem ser diferentes cada vez que a ferramenta de programação genética, no caso a Chameleon, for executada, enquanto que com os operadores de mutação, que fazem uma alteração sintática simples no programa em teste, as alterações deverão sempre ser as mesmas, em dependência dos operadores selecionados.
- Alternativas Equivalentes: devido às possibilidades fornecidas pela Proteum em relação às alterações sintáticas no código do programa em teste, o número de alternativas que necessitam de análise para verificação da possível equivalência foi o que demandou a necessidade de mais esforço por parte do testador na identificação dos equivalentes manualmente. Já com relação a GPTesT, o número de alternativas que precisaram ser analisadas foi bem menor, e conseqüentemente o esforço também.
- Número de Casos de Teste: para nenhum dos programas o número de casos de teste efetivo foi menor para a Proteum. A média para a Proteum foi 14,25, enquanto a média de casos de teste efetivos para a GPTesT foi 5,5. Observe a Tabela 6.4, o que demonstra, também, um menor custo.

Tabela 6.4: Casos de Teste Efetivos.

Problema	GPTesT	Proteum
Fatorial	5	5
Maior de Três números	6	26
MMC	4	14
MDC	7	12

- O tempo de execução das alternativas não pode ser considerado porque os experimentos foram realizados em máquinas diferentes e porque a GPTesT fornece como resultado somente o tempo de execução das alternativas, que deveria ser somado ao tempo de geração das mesmas pela Chameleon; e a Proteum fornece o tempo em que as alternativas e o programa em teste foi executado.

Então, quanto a custos, o esforço exigido do testador foi maior com o uso da ferramenta Proteum, no momento de geração de casos de teste e de análise das alternativas equivalentes, mas deve-se salientar que o esforço do testador na aplicação da GPTesT, é maior para a geração das alternativas, devido a definição da gramática e do conjunto de casos de teste que deve traduzir o problema, por causa da Chameleon, uma ferramenta de PG orientada a gramática.

Sobre o *strength* da abordagem baseada em programação genética em relação a análise de mutantes, seria necessário que os conjuntos de casos de teste utilizados no experimento fossem os mesmos tanto para a GPTesT quanto para a Proteum para que se possa analisar isto.

Com relação à eficácia obtida com o conjunto de casos de teste, pode-se dizer que tanto a GPTesT quanto a Proteum revelaram, durante a atividade de teste, o defeito existente no programa fatorial inicialmente testado. Porém esse fator, assim como, o fator *strength* serão objetos de estudos futuros.

7 Conclusões e Trabalhos Futuros

Neste trabalho é apresentada uma nova abordagem para o teste de programas chamada Teste Baseado em Programação Genética (GPBT) e a ferramenta, *Genetic Programming Based Testing Tool* (GPTesT), que permite a sua aplicação. Foi considerado, principalmente, o critério de teste denominado Análise de Mutantes, pois a GPBT é similar a esse Critério, porém os conceitos de evolução são usados ao invés de operadores de mutação.

As alternativas são geradas geneticamente, o que permite que sejam geradas alternativas que não diferem do programa em teste *P* por uma simples alteração sintática, como é feito no critério Análise de Mutantes através de operadores de mutação. O pressuposto do efeito do acoplamento não é necessário, de modo que o número de alternativas pode ser menor e podem ocorrer interações entre defeitos, sendo revelados, deste modo, outros tipos de defeitos, em relação aos revelados pelo uso de operadores de mutação.

O trabalho descreve dois procedimentos de teste para auxiliar o testador nas tarefas de seleção e avaliação de um conjunto de casos de teste, da mesma forma que um critério de teste. Na seleção de casos de teste, é considerado o conjunto de casos de teste usado no processo de geração das alternativas, com a ferramenta de programação genética, o que é uma vantagem, pois reduz o número de possíveis alternativas anômalas e vivas, e por conseqüência, os esforços gastos no teste.

Outra vantagem importante da GPBT é a independência de uma linguagem ou paradigma de programação, o que a torna muito mais flexível. A geração e seleção de dados de teste não estão baseadas no programa sendo testado, e sim nos casos de teste iniciais para a ferramenta de PG, que podem ser gerados baseando-se na especificação. Isso minimiza os problemas relacionados a

caminhos ausentes e traz benefícios para a fase de manutenção, bem como permite a combinação da abordagem GPBT com técnicas de teste funcionais.

A programação genética também pode ser utilizada para o teste de comparação, mencionado no Capítulo 2. Esse tipo de teste gera diferentes versões para *P* e é aplicado em casos onde a confiabilidade é crítica e existe redundância de software. A geração manual de diferentes versões para *P*, requer muito esforço e, geralmente, não tem sido aplicada em outros casos. A utilização da PG pode reduzir esse custo.

Neste trabalho a ferramenta GPTesT auxilia o teste de programas escritos na linguagem C e interage com a ferramenta Chameleon para a geração das alternativas por programação genética orientada a gramática. A implementação de GPTesT permite futuras extensões, uma delas é a geração de alternativas usando outras ferramentas que evoluam programas escritos em outras linguagens ou paradigmas, como por exemplo, em Lisp. Além disso, atualmente são usados programas simples para evolução genética e por conseqüência no experimento realizado com a GPTesT, portanto, pretende-se conduzir experimentos futuros com a aplicação da ferramenta em problemas mais complexos e avaliar outros fatores, tais como, eficácia e *strength* com relação a Proteum e PokeTool.

Da mesma forma que outras ferramentas de teste, GPTesT possui algumas limitações. Isto ocorre devido as limitações inerentes a atividade de teste, que impedem a completa automatização dessa atividade. Mas, no futuro é uma pretensão implementar mecanismos na GPTesT que permitam a redução dessas limitações. Os mecanismos seriam dados por heurísticas para determinar alternativas equivalentes e algoritmos genéticos para gerar automaticamente dados de teste.

Outro fator que deve ser considerado é que GPTesT, utilizando a ferramenta Chameleon, ou outra ferramenta que providencia programação genética, pode ter como alternativas geradas, em duas ou mais execuções, possibilidades diferentes, mesmo que a configuração inicial seja a mesma. Isto pode ser considerado para a

avaliação de um conjunto de casos de teste, e deve ser aplicado em experimentos futuros como fator para comparar esses conjuntos.

A utilização da GPBT e da GPTesT restringe-se a certos tipos de problemas, devido às limitações impostas pelas ferramentas de PG existentes. Entretanto, a Computação Evolucionária é um campo emergente, alvo de novos estudos de pesquisa, o que torna a proposta descrita nesse trabalho bastante promissora. A ferramenta Chameleon, atualmente, está sendo estendida para evoluir funções [42], e possivelmente, será ampliada para executar a evolução de classes. Isso permitirá que, no futuro, a GPTesT possa apoiar o teste de integração e também o teste de programas em C++.

Um próximo trabalho a ser conduzido deverá explorar Computação Evolucionária, mais particularmente o ramo de Programação Evolucionária, no teste de especificações baseadas em máquinas de estado finito e diagramas de estado, aproveitando as idéias aqui apresentadas.

Referências Bibliográficas

[1] BANZHAF, W. et al. **Genetic Programming: An Introduction**. Morgan Kaufmann Publishers, 1998.

[2] BERGADANO, F., GUNETTI, D. **Inductive Logic Programming: from Machine Learning to Software Engineering**. MIT Press, 1995.

[3] BRILLIANT, S. S. and KNIGHT, J.C. and LEVENSON, N. G. **The Consistent Comparison Problem in N-Version Software**. ACM Software Engineering Notes, vol. 12, n. 1, p. 29-34, January, 1987.

[4] BUDD, T. A. **Mutation Analysis: Ideas, Examples, Problems and Prospects, Computer Program Testing**. North-Holand Publishing Company, 1981.

[5] BUDD, T., ANGLUIN, D. **Two notions of correcteness and their relation to testing**. Acta Informatica, vol. 18(1):31-45, November, 1982.

[6] CHAIM, M. L. **POKE-TOOL – Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseados em Análise de Fluxo de Dados**. Tese de Mestrado, DCA/FEE/UNICAMP, Campinas, SP, Abril, 1991.

[7] CHOW, T. S. **Testing Software Design Modeled by Finite-State Machines**. IEEE Trans. On Soft. Engin., vol 4, n. 3, p. 178 -187, 1978.

[8] DARWIN, C. **On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life**. Murray, London, UK, 1859.

[9] DELAMARO, M. E. **Mutação de Interface: Um Critério de Adequação Interprocedimental para o Teste de Integração**. Tese de Doutorado, Instituto de Física de São Carlos – USP, junho, 1997.

[10]DELAMARO, M. E. **Proteum – Um Ambiente de Teste Baseado na Análise de Mutantes**. Dissertação de Mestrado - C, ICMSC/USP: julho, 1996.

[11]DEMILLO, R. A. & OFFUTT, J. **Constrained-Based Automatic Test Data Generation**. IEEE Transactions on Software Engineering, v. 17, n. 9, September, 1991.

[12]DEMILLO, R. A. **Hints on Test Data Selection: Help for the Practicing Programmer**. IEEE Computer, April, 1978.

[13]DEMILLO, R. A and GWIND, D. C. and KING, K. N. **An Extended Overview of the Mothra Software Testing Environment**. Proc. Of the Second Workshop on Software Testing, Verification and Analysis, Banff – Canada, Computer Science Press, july 19-21, p. 142-151, 1988.

[14]DEMILLO, R. A and OFFUTT, A. J. **Experimental Results on Automatic Test Case Generation**. ACM Transactions on Software Engineering and Methodology, vol. SE-2, n 2, p. 109-127, april, 1993.

[15]FABBRI, S.C.P,F. and et al. **Aplicação do Critério Análise de Mutantes na Validação de Especificações Baseadas em Statecharts**. XI Simpósio Brasileiro de Engenharia de Software, Fortaleza – Ceará, outubro, 1997.

[16]FOGEL, M. **Evolutionary Computation - Principles and Practice for Signal Processing**. Bellinghan -SPIE Press, 2000.

[17]FRANKL, F. G. **The use of Data Flow Information for the Selection and Evaluation of Software Test Data**. PhD Thesis, Department of Computer Science, New York University, October, 1987.

[18]FRASER, A. P. **Genetic Programming in C++**. University of Salford, Cybernetics Research Institute, Techinical Report 040, 1994.

[19]FUJIWARA, S. and et al. **Test Selection Based on Finite State Models**. IEEE Trans. On Soft. Engin., vol 17, n 6, june, 1991.

[20]HOLLAND, J.H. **Adaptation in Natural and Artificial Systems**. University of Michigan Press, Second Edition, MIT Press, 1992.

[21]HORGAN, J. R, LONDON, S. **ATAC – Automatic Test Coverage Analysis for C Programs**. Bellcore International Memorandum, June, 1990.

[22]HOWDEN, W. **Weak mutation testing and completeness of test sets**. IEEE Transactions on Software Engineering, vol. SE-8(4):371-379, July, 1982.

[23] **IEEE Standard Glossary of Software Engineering Terminology**. Padrão 610.12, IEEE, 1990.

[24]KING, K. N. & OFFUTT, A. Jefferson. **A Fortran Language System for Mutation-Based Software Testing**. Symposium on Interpreters and Interpretive Techniques, St. Paul MN, 1987.

[25]KNIGHT, J. and AMMANN, P. **Testing Software Using Multiple Versions**. Relatório n. 89029N, Software Productivity Consortium, Reston, Virginia, june, 1989.

[26]KOZA, J. R. **Genetic Programming: On the Programming of Computers by Means of Natural Selection**. MIT Press, 1992.

[27]KOZA, J. R.; **Genetic Programming II: Automatic Discovery of Reusable Programs**. MIT Press, 1994.

[28]MALDONADO, J. C, CHAIM, M., JINO, M, **Briding the gap in the presence of infeasible paths: Potencial uses testing criteria**. In XII International Conference of he Chilean Science Computer Society, p. 323-340, Santiago, Chile, October, 1992.

[29]MALDONADO, J. C. **Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software**. Tese de Doutorado, DCA/FEE/UNICAMP. Campinas, SP, julho, 1991.

[30]MALDONADO, José Carlos, et al. **Aspectos Teóricos e Empíricos de Teste de Cobertura de Software**. Notas Didáticas do Instituto de Ciências Matemáticas de São Carlos, nº 31, São Carlos: Jun, 1998.

[31]MATHUR, A. P. & WONG, W. E. **An Empirical Comparison of Data Flow and Mutation-Based Test Adequacy Criteria**. Software Testing, Verification and Reliability, v. 4(1):9-31, March, 1994.

[32]MCCABE, T. **A Software Complexity Measure**. IEEE Trans. Software Engineering, v. 2, December 1976, pp. 308-320.

[33]MICHALEWICZ, Z., MICHALEWICZ, M. **Evolutionary Computation Techniques and Their Applications**. In International Conference on Intelligent Processing Systems, p. 14-25, China, October, 1997.

[34]MITCHELL, Melanie. **An Introduction to Genetic Algorithms.** Bradford Book, 1996.

[35]MYERS, G. **The Art of Software Testing**. Wiley, 1979.

[36]MORELL, L. J. **Theoretical insights into fault-based testing**. In Proc. of Workshop on Software Testing, Verification and Analysis, p. 45-62, Banff, Canada, 1988.

[37]OFFUTT, A. J. **The coupling effect: Fact or Fiction?** In Proc. Of Workshop on Software Testing, Verification and Analysis, pages 131-140. 1989.

[38]PRESSMAN, Roger S.. **Software Engineering - A Practitioner's Approach**. 5ª ed., McGraw-Hill, New York, 2000.

[39]RAPPS, S. & WEYUKER, E. **Data Flow analysis techniques for test data selection**. In Proc. of International Conference on Software Engineering, Tokio – Japan, September, 1982.

[40]RAPPS, S. & WEYUKER, E. J. **Selecting Software Test Data Using Data Flow Information**. IEEE Transactions on Software Engineering, SE-11(4), April, 1985.

[41]RAMAMOORTHY, C. and SIU-BUN, F. H. and CHEN, W. **On automated generation of program test data**. IEEE Transactions on Software Engineering, vol. SE-2(4):293-300, December, 1976.

[42] RODRIGUEZ, E. I. M. **Evolução de Funções em Programação Genética Orientada a Gramática**. Dissertação de Mestrado, Departamento de Informática, UFPR, em elaboração.

[43]SPINOSA, E. et al. Chameleon: **A Generic Tool for Genetic Programming**. Encontro Nacional de Inteligência Artificial, ENIA 2001, XXI Congresso da Sociedade Brasileira de Computação - SBC,.Fortaleza, Ceará, Agosto, 2001.

[44]VERGÍLIO, S. R. **Critérios Restritos de Teste de Software: Uma Contribuição para Gerar Dados de Teste mais Eficazes**. Doctorate Dissertation, DCA/FEEC/Unicamp, Campinas – SP, Brazil, July, 1997.

[45]VERGÍLIO, S., MALDONADO, J., JINO, M. **Infeasible paths within the context of data flow based criteria**. In VI International Conference on Software Quality, p. 310-321. Ottawa-Canada, October, 1996.

[46]WEINBRENNER, T. **Genetic Programming Kernel version 0.5.2 – C++ Class Library**. http://thor.emk.e-technik.tu-darmstadt.de/~thomasw/gpkernel_toc.html, 1997.

[47]WEYUKER, E. **Assessing test data adequacy through program inference**. ACM Trans. On Programming Languages and Systems, vol. SE-5(4):641-655, 1983.

[48]WEYUKER, E., WEISS, S. N., HAMLET. **Comparison of program testing strategies**. In Proc. Of the Fourth Symposium on Software Testing, Analysis and Verification, p. 154-164, ACM Press, Victoria – Canada, 1991.

[49]WHIGHAM, P. A.; **Grammatical Bias for Evolutionary Learning**. PhD thesis. School of Computer Science, University of New South Wales, Australian Defense Force Academy, 1996

[50]WONG, M. L., LEUNG, K. S. **Data Minig Using Grammar Based Genetic Programming and Applications.** Kluwer Academic Publishers, 2000.

[51]WONG, W. E. et al. **Mutation versul All-Uses: An Empirical Evaluation of Cost, Strength and Effectiveness**. In Software Quality and Productivity – Theory, Practice, Education and Training. Hong Kong, December, 1994.

[52]ZONGKER, D., PUNCH, B. et al. **Lil-gp 1.01 User's Manual**. Michigan State University, 1996.

Apêndice A. Programas utilizados no Experimento com a GPTesT

Programa Fatorial (Figura A.1)→ a função fatorial de um número natural n é o produto de todos os n primeiros números naturais, isto é, Fat(n)=n!=1.2.3.4....n. Sendo que, considera-se Fat(0)=1.

```
/*calcula o fatorial de um numero*/

#include <stdio.h>
#include <stdlib.h>

main (int argc, char *argv[])
{
 float x, fat;
 int i;
 fat =1;

 if (argc > 1)
   x = atof(argv[1]);
 else
   printf("ERRO... - sintaxe: fatorial [numero]");

 if (x == 0)
   printf("1\n");
 else {
     for(i=x; i>0; i--)
       fat = fat * i;
     printf("%f\n", fat);
 }
}
```

Figura A.1: Código do Programa Fatorial.

Programa Maior de Três Números (Figura A.2) → Dados três números inteiros, a função fornece como resposta o maior entre eles.

```
/* Calcula o maior número entre 3 números de entrada */

# include <stdio.h>
# include <stdlib.h>

main(int argc, char * argv[])
{
  int a, b, c;
 if (argc != 4)
 {
   printf("Erro na quantidade de parâmetros. Uso: maior <num1>
<num2> <num3>\n");
    exit(1);
 }
 a = atoi(argv[1]);
 b = atoi(argv[2]);
 c = atoi(argv[3]);
 if ( (a>=b)&&(a>=c) )
    printf("%d\n", a);
 else if ( (b>=c)&&(b>=a) )
    printf("%d\n", b);
 else if ( (c>=a)&&(c>=b) )
    printf("%d\n", c);
}
```

Figura A.2: Código do Programa Maior de Três Números.

Programa Mínimo Múltiplo Comum – MMC (Figura A.3) → Dados dois números inteiros positivos a e b não nulos, calcula-se o mínimo múltiplo comum como sendo o menor inteiro positivo, múltiplo comum de a e b.

```
/*calcula o mmc entre a e b
# include <stdio.h>
# include <stdlib.h>

main(int argc, char * argv[])
{
  int a, b, resto, A, B; /* a e b contêm os valores iniciais */
if (argc != 3)
 {
         printf("Erro na quantidade de parâmetros. Uso: mmc
<num1> <num2> \n");
         exit(1);
 }
 a = atoi(argv[1]);
 b = atoi(argv[2]);
 A = a;
 B = b;
 if ((a > 0) && (b > 0)){
        do
        {
               resto = A % B;
               A = B;
               B = resto;
        }
        while (resto != 0);
    printf("%d", (a*b)/A);
 }
 else
 {
  printf("0\n");
 }
}
```

Figura A.3: Código do Programa MMC.

Programa Máximo Divisor Comum – MDC (Figura A.4) → Dados dois números inteiros positivos a e b não nulos, o máximo divisor comum é o maior inteiro que divide simultaneamente a e b.

```
/* calcula o mdc entre a e b

# include <stdio.h>
# include <stdlib.h>

main(int argc, char * argv[])
{
 int a, b, resto;
  if (argc != 3)
 {
        printf("Erro na quantidade de parâmetros. Uso: mdc
<num1> <num2> \n");
        exit(1);
 }
 a = atoi(argv[1]);
 b = atoi(argv[2]);
  if (( b > 0) && (a > 0))
   do
    {
     resto = a % b;
     a = b;
     b = resto;
    }
   while (resto != 0); /* a irá conter o mdc */
  else
   a = 0;
  printf("%d\n", a);
}
```

Figura A.4: Código do Programa MDC.

Apêndice B. Procedimento com resultado de cada passo realizado com os programas no emprego de GPTesT

Programa Fatorial

- Primeiro passo: executar a ferramenta Chameleon para gerar as alternativas para o cálculo do fatorial de um número.

Para executar a Chameleon foram utilizados como parâmetros: os casos de teste (entrada e saída) da Tabela B.1, a configuração inicial *I* da Figura B.1, o número de rodadas igual a 10 e o número de gerações igual a 100.

Tabela B.1: Casos de Teste para o Programa Fatorial.

Entrada	Saída
X	Y
0	1
1	1
2	2
3	6
4	24
5	120

```
[begin]
[parameters]
population size = 500
tournament size = 7
maximum depth for initial random programs = 30
maximum depth for programs created during the run = 60
crossover rate = 90
mutation rate = 0
elitist = N
threshold = 0.01
[compiler]
cl -nologo -G6 -MT -Fepop.exe
[result-producing branch]
terminal set = {X}
function set = {MT,LT,MUL,SUB}
output variable = Y
[result-producing branch productions]
<code>  -> <def> <prog>
<def>   -> Y = <cte>;
<prog>  -> while(<expb>) { <bloco> }
<bloco> -> <exp>
<bloco> -> <bloco> <exp>
<exp>   -> <var> = <op>(<var>,<var>);
<exp>   -> <var> = <op>(<var>,<cte>);
<expb>  -> <opc>(<var>,<cte>)
<opc>   -> MT
<opc>   -> LT
<op>    -> MUL
<op>    -> SUB
<var>   -> X
<var>   -> Y
<cte>   -> 1
<cte>   -> 0
[fitness cases]
source -> fatorial.dat
[end]
```

Figura B.1: Configuração Inicial I para o Programa Fatorial.

Para exemplificar os resultados encontrados pela Chameleon para o programa fatorial, na Tabela B.2 podem ser visualizadas as melhores alternativas selecionadas em cada uma das dez rodadas executadas pela ferramenta.

Tabela B.2: Exemplos de Alternativas Geradas por Chameleon.

Rodada	Melhor Alternativa Selecionada
1	Y =1;while(MT(Y,0)) {Y =SUB(X,Y);X =MUL(Y,Y);Y =MUL(Y,Y);Y =MUL(Y,Y);Y =MUL(Y,Y);Y =MUL(X,Y); }
2	Y =1;while(MT(X,1)) {Y =MUL(X,X);X =MUL(Y,Y);X =SUB(Y,X);Y =MUL(Y,X);Y =SUB(Y,X);X =MUL(Y,Y); }
3	Y =1;while(MT(X,1)) {X =MUL(X,X);X =SUB(X,Y);Y =MUL(Y,1);Y =MUL(Y,0);Y =SUB(Y,X);Y =MUL(X,Y); }
4	Y =1;while(MT(X,0)) {Y =MUL(X,Y);X =SUB(X,1); }
5	Y =1;while(MT(X,1)) {X =MUL(Y,X);Y =SUB(Y,1);Y =MUL(X,1);Y =MUL(X,Y);X =SUB(Y,X);Y =MUL(X,1); }
6	Y =1;while(MT(X,0)) {Y =MUL(Y,Y);Y =MUL(X,X);Y =MUL(X,Y);Y =SUB(X,Y);X =SUB(X,Y);X =MUL(Y,X);Y =SUB(X,Y);Y =MUL(Y,X);X =MUL(X,X); }
7	Y =1;while(MT(X,0)) {Y =MUL(X,X);X =MUL(Y,Y);Y =SUB(Y,Y);Y =MUL(X,X);Y =SUB(Y,X);Y =MUL(Y,X);X =SUB(Y,1);X =MUL(Y,Y);Y =MUL(Y,1);X =SUB(Y,1);Y =SUB(Y,Y);Y =MUL(Y,Y);Y =MUL(Y,X); }
8	Y =1;while(MT(X,0)) {Y =SUB(Y,X);X =SUB(X,Y);Y =SUB(Y,Y); }
9	Y =1;while(MT(X,1)) {Y =MUL(X,Y);Y =MUL(X,X);Y =MUL(X,1);X =SUB(Y,1);Y =MUL(Y,1);Y =MUL(X,0);Y =SUB(Y,X);X =SUB(Y,Y);X =MUL(Y,1);X =MUL(Y,X); }
10	Y =1;while(MT(X,1)) {X =SUB(Y,X);Y =MUL(X,X);Y =MUL(X,X);Y =MUL(X,Y);Y =MUL(X,X);X =MUL(X,Y);X =MUL(X,X); }

- Segundo passo: gerar os casos de teste (entradas) que serão utilizados para a execução do programa em teste e das alternativas, com a finalidade de comparar seus resultados, para poder identificar alternativas vivas, mortas, anômalas ou equivalentes. Os dados de teste selecionados pelo testador estão na Tabela B.3.

Tabela B.3: Dados de Teste para a Gptest – Programa Fatorial.

Número	Entrada
	X
1	1
2	0
3	2
4	4
5	3
6	6
7	5
8	7

➢ Terceiro passo: utilizar os resultados da Chameleon (arquivos que contém todas as alternativas geradas), a configuração inicial *I* (Figura B.1), o programa em teste (Figura A.1) e os dados de teste estabelecidos pelo testador (Tabela B.3) para executar a GPTesT.

A execução da GPTesT, com os parâmetros mencionados acima, gerou resultados conforme a seqüência exposta a seguir.

☑ Verificação de programas gerados idênticos pela Chameleon...

Foram geradas 814 alternativas de programas, sendo identificadas 402 alternativas diferentes. As alternativas diferentes que permanecem na execução da GPTesT como alternativas vivas.

☑ Execução do programa em teste com os casos de teste, gerando os resultados (a saída respectiva)...

Tabela B.4: Resultados da Execução do Programa em Teste com os Casos de Teste

Número	Entrada	Saída
	X	Y
1	1	1
2	0	0
3	2	2
4	4	24
5	3	6
6	6	720
7	5	120
8	7	5040

☑ Execução das Alternativas, verificando o estado da alternativa e alterando-o quando o resultado gerado não é válido, devido a alguma anomalia ou quando o resultado gerado é diferente do obtido com o programa em teste para o mesmo caso de teste.

A seguir na Tabela B.5 é apresentado o número de alternativas mortas após a execução de cada caso de teste. As alternativas que são consideradas mortas não são executadas com o próximo caso de teste.

Tabela B.5:Número de Alternativas Mortas com a Execução dos Casos de Teste.

Número	Entrada	Número de Alternativas Mortas
	X	
1	1	105
2	0	2
3	2	281
4	4	13
5	3	0
6	6	0
7	5	0
8	7	0

O tempo de execução das alternativas geradas com os casos de teste especificados pelo testador foi de 00:02:28h (dois minutos e vinte e oito segundos).

- Quarto passo: verificar a Medida de Cobertura, pois em dependência do resultado desejado pelo testador, a execução da GPTesT pode ser encerrada ou não.

```
Total de Alternativas: 402
Total de Alternativas Anômalas: 0
Total de Alternativas Vivas: 1
Total de Alternativas Equivalentes: 0
Total de Casos de Teste: 8

MEDIDA DE COBERTURA: 0.997512
```

Figura B.2: Resultado Calculado para a Medida de Cobertura.

Observando o resultado do cálculo da Medida de Cobertura, verifica-se que há uma alternativa viva, ou seja, que é considerada solução para o problema do fatorial de um número. Esta alternativa é apresentada na Tabela B.6. Portanto, o testador deverá decidir se: considera o resultado satisfatório, inclui novos casos de teste, identifica se a alternativa restante é equivalente, ou ainda, devido aos resultados obtidos com o programa em teste, detecta alguma falha em seu código e corrige-a, realizando o teste novamente.

Tabela B.6: Alternativa para o Cálculo do Fatorial.

Número	Alternativa Gerada Viva
120	Y =1;while(MT(X,0)) {Y =MUL(X,Y);X =SUB(X,1); }

A opção foi incluir mais um caso de teste, cuja entrada é X = -1.

Este caso de teste não alterou o resultado visto na Figura B.2, pois o programa em teste e a alternativa geraram como resultado o valor 1. Porém, este resultado foi considerado errado, conforme a definição do fatorial de um número:

→ A função fatorial de um número natural n é o produto de todos os n primeiros números naturais, isto é, Fat(n)=n!=1.2.3.4....n. Tomaremos Fat(0)=1.

Dessa forma, percebeu-se que existia um erro no código do programa em teste. O mesmo foi corrigido, conforme apresentado na Figura B.3. E o procedimento de teste voltou ao ponto em que ocorre a execução dos casos de teste com o programa em teste.

```
/*calcula o fatorial de um numero*/
#include <stdio.h>
#include <stdlib.h>
main (int argc, char *argv[])
{
 float x, fat;
 int i;
 fat =1;

 if (argc > 1)
   x = atof(argv[1]);
 else
   printf("ERRO... - sintaxe: fatorial [numero]");

 if (x == 0)
   printf("1\n");
else
  if (x<0)
    printf("0\n"); //valor invalido
  else {
     for(i=x; i>0; i--)
       fat = fat * i;
     printf("%f\n", fat);
  }
}
```

Figura B.3: Código do Programa Fatorial Alterado.

- Execução do programa em teste.

Tabela B.7: Resultados da Execução do Programa em Teste Alterado.

Número	Entrada	Saída
	X	Y
1	1	1
2	0	0
3	2	2
4	4	24
5	3	6
6	6	720
7	5	120
8	7	5040
9	-1	0

- Execução das Alternativas.

A seguir na Tabela B.8 é apresentado o número de alternativas mortas após a execução de cada caso de teste. As alternativas que são consideradas mortas não são executadas com o próximo caso de teste.

Tabela B.8: Número de Alternativas Mortas na Execução.

Número	Entrada	Número de Alternativas Mortas
	X	
1	1	105
2	0	2
3	2	281
4	4	13
5	3	0
6	6	0
7	5	0
8	7	0
9	-1	1

- Verificação da Medida de Cobertura.

```
Total de Alternativas: 402
Total de Alternativas Anômalas: 0
Total de Alternativas Vivas: 0
Total de Alternativas Equivalentes: 0
Total de Casos de Teste: 9

MEDIDA DE COBERTURA: 1
```

Figura B.4: Resultado da Medida de Cobertura.

O resultado obtido demonstrou que a alternativa que permanecia viva antes da alteração do código foi morta com o caso de teste X = -1. A Medida de Cobertura com resultado igual a 1, portanto o procedimento de teste pôde ser encerrado.

Nas Tabelas B.9 e B.10 são apresentados os resultados obtidos com o programa fatorial, de um modo geral, e todos os casos de teste utilizados, respectivamente.

A Tabela B.9 possui as colunas: Primeiro, Segundo e Final que se relacionam ao número de vezes que o GPTesT foi executado com novos casos de teste ou com a identificação de alternativas equivalentes. Nas linhas são fornecidos: o número total de alternativas geradas pela Chameleon; de alternativas diferentes ou ativas, que são consideradas para a execução da ferramenta; de alternativas anômalas, de alternativas vivas, de alternativas equivalentes, de casos de teste usados nas execuções até o momento, o tempo de execução e os resultados da medida de cobertura em cada execução da ferramenta.

Na Tabela B.10 estão disponíveis os casos de teste usados e o número de alternativas mortas com cada um deles.

Tabela B.9: Resultados Programa Fatorial.

Resultado	Primeiro	Segundo	Final[1]
Nº Alternativas Geradas			814
Nº de Alternativas Diferentes			402
Nº Alternativas Anômalos	0	0	0
Nº Alternativas Vivas	1	1	0
Nº Alternativas Equivalentes	0	0	0
Nº Casos Teste	8	9	9
Casos Teste			Tabela 6.3
Tempo de Execução	00:02:32h	00:00:15h	00:02:47
MEDIDA DE COBERTURA	0.997512	0.997512	1

Tabela B.10: Casos de Teste para o Programa Fatorial.

Número do Caso de Teste	Caso de Teste	Número de Alternativas Mortas
	X	
1	1	105
2	0	2
3	2	281
4	4	13
5	3	0
6	6	0
7	6	0
8	7	0
9	-1	1

Programa Maior de Três Números

- Primeiro passo: Executar a Chameleon, gerando as alternativas para a GPTesT.

Os parâmetros utilizados para executar a Chameleon foram:

---

[1] O programa para calcular o fatorial de um número foi alterado. Todos os casos de teste foram executados novamente.

- casos de teste (Tabela B.11);
- configuração inicial *I* (Figura B.5);
- número de rodadas, 10;
- número de gerações, 100.

Tabela B.11: Casos de Teste.

Entrada			Saída
A	**B**	**C**	**Z**
1	2	3	3
1	3	2	3
2	1	3	3
2	3	1	3
3	1	2	3
3	2	1	3
15	2	17	17
20	1	0	20

```
[begin]
[parameters]
population size = 500
tournament size = 7
maximum depth for initial random programs = 30
maximum depth for programs created during the run = 60
crossover rate = 90
mutation rate = 0
elitist = N
threshold = 0.01
[compiler]
cl -nologo -G6 -MT -Fepop.exe
[result-producing branch]
terminal set = {A,B,C}
function set = {MT,LT}
output variable = Z
[result-producing branch productions]
<code>  -> <stmt>
<stmt>  -> if(<expb>) {<stmt>}
<stmt>  -> if(<expb>) {<stmt>} else {<stmt>}
<stmt>  -> Z = <var>;
<expb>  -> <op>(<var>,<var>)
<op>    -> MT
<op>    -> LT
<var>   -> A
<var>   -> B
<var>   -> C
[fitness cases]
source -> maior.dat
[end]
```

Figura B.5: Configuração Inicial I.

Como exemplo de alternativas geradas pela ferramenta Chameleon, podem ser visualizadas algumas das alternativas na Tabela B.12.

Tabela B.12: Alternativas Geradas para o Problema Maior de Três Números.

Rodada	Alternativa
1	if(MT(B,A)) {if(LT(C,B)) {if(MT(B,B)) {if(LT(B,A)) {if(LT(A,C)) {Z =B;}} else {Z =A;}} else {Z =B;}} else {if(LT(A,C)) {Z =C;} else {Z =A;}}} else {if(MT(C,A)) {Z =C;} else {Z =A;}}
2	if(LT(A,C)) {Z =C;} else {if(MT(B,A)) {if(LT(C,C)) {if(MT(A,B)) {if(MT(A,A)) {if(MT(B,B)) {Z =C;}}}} else {Z =B;}} else {if(LT(B,A)) {if(LT(C,A)) {Z =A;} else {if(LT(A,B)) {if(MT(B,A)) {if(LT(A,A)) {Z =C;} else {Z =B;}}}}}}}
3	if(LT(B,B)) {if(MT(B,B)) {if(MT(B,A)) {if(MT(C,C)) {if(MT(C,C)) {if(LT(B,B)) {if(LT(C,A)) {if(LT(A,A)) {if(LT(B,B)) {if(LT(C,A)) {if(LT(C,A)) {Z =A;}}} else {if(MT(A,A)) {if(LT(C,B)) {if(MT(C,C)) {if(MT(A,C)) {if(LT(B,C)) {Z =A;}}}}}}}} else {if(LT(B,B)) {if(LT(B,B)) {if(MT(C,B)) {Z =B;}} else {Z =C;}} else {if(LT(A,C)) {Z =A;}}}}} else {Z =B;}} else {Z =B;}} else {Z =A;}} else {if(LT(B,B)) {if(LT(B,A)) {if(MT(C,B)) {if(LT(C,A)) {if(LT(C,A)) {Z =A;}}}} else {Z =C;}} else {if(MT(C,A)) {if(MT(B,C)) {if(LT(C,A)) {if(MT(C,B)) {if(MT(A,B)) {if(LT(A,C)) {if(MT(A,B)) {Z =B;} else {Z =A;}} else {if(MT(B,A)) {Z =B;}}} else {if(LT(C,C)) {if(LT(B,C)) {Z =A;}}}}} else {Z =B;}} else {Z =C;}} else {if(MT(A,B)) {if(MT(C,B)) {if(MT(A,C)) {Z =A;} else {Z =C;}} else {Z =A;}} else {Z =B;}}}}
4	if(LT(C,B)) {if(LT(A,B)) {Z =B;} else {Z =A;}} else {if(LT(A,C)) {Z =C;} else {Z =A;}}
5	if(MT(C,B)) {if(MT(A,C)) {if(LT(B,C)) {if(LT(B,A)) {if(LT(B,A)) {Z =A;}}}} else {Z =C;}} else {if(LT(B,A)) {Z =A;} else {Z =B;}}
6	if(LT(A,B)) {if(LT(C,B)) {if(MT(C,C)) {if(LT(B,A)) {if(MT(A,C)) {Z =C;}} else {Z =C;}} else {Z =B;}} else {Z =C;}} else {if(LT(A,C)) {if(LT(C,C)) {if(MT(C,C)) {if(MT(B,A)) {if(LT(A,A)) {Z =B;}} else {if(MT(B,A)) {Z =A;}}} else {Z =B;}} else {Z =C;}} else {Z =A;}}
7	if(LT(A,C)) {if(MT(C,B)) {Z =C;} else {if(LT(B,A)) {Z =A;} else {Z =B;}}} else {if(LT(B,A)) {Z =A;} else {Z =B;}}
8	if(LT(B,C)) {if(MT(A,C)) {if(MT(A,A)) {Z =B;} else {if(MT(A,B)) {if(LT(B,B)) {Z =A;} else {Z =A;}} else {Z =B;}}} else {Z =C;}} else {if(MT(A,B)) {if(MT(B,B)) {Z =A;} else {if(LT(C,B)) {Z =A;}}} else {if(LT(C,B)) {if(LT(A,B)) {Z =B;}}}}
9	if(MT(B,C)) {if(LT(C,A)) {if(MT(A,B)) {Z =A;} else {Z =B;}} else {Z =B;}} else {if(LT(A,C)) {Z =C;} else {Z =A;}}
10	if(MT(A,C)) {if(MT(B,A)) {Z =B;} else {if(MT(C,A)) {Z =C;} else {Z =A;}}} else {if(MT(C,B)) {Z =C;} else {if(LT(A,C)) {Z =B;} else {Z =C;}}}

➢ Segundo passo: gerar os dados de teste para a GPTesT ser executada.

Tabela B.13: Dados de Teste para Gptest – Programa Maior de Três Números.

Número	Entrada		
	A	B	C
1	0	1	2
2	1	2	0
3	2	0	1
4	2	1	0
5	1	0	2
6	0	2	1
7	0	1	1
8	1	1	0
9	1	0	1
10	0	2	0
11	0	0	2
12	2	0	0
13	2	2	2

➢ Terceiro passo: Executar a GPTesT utilizando os resultados gerados pela Chameleon; a configuração inicial *I* (Figura B.5); o programa em teste (Figura 5.9) e os dados de teste (Tabela B.13).

Os resultados obtidos com a execução da GPTesT estão apresentados a seguir.

- o Verificação de programas gerados idênticos pela Chameleon.

  Foram geradas 170 alternativas, sendo identificadas 115 como diferentes. O número baixo de alternativas geradas se deve ao fato de em todas as rodadas terem sido encontradas possíveis soluções para o problema. É bom lembrar que um dos critérios de parada de busca por solução na programação genética é a obtenção de uma possível solução, isto quer dizer que quando uma é encontrada, outra rodada é iniciada.

- o Execução do programa em teste.

Tabela B.14: Resultados Obtidos com a Execução do Programa Maior de Três Números.

Número	Entrada			Saída
	A	B	C	Z
1	0	1	2	2
2	1	2	0	2
3	2	0	1	2
4	2	1	0	2
5	1	0	2	2
6	0	2	1	2
7	0	1	1	1
8	1	1	0	1
9	1	0	1	1
10	0	2	0	2
11	0	0	2	2
12	2	0	0	2
13	2	2	2	2

- Execução das Alternativas. Na Tabela B.15 são apresentados os casos de teste e o número de alternativas mortas com cada um. O tempo de execução das alternativas foi 00:02:15h (dois minutos e quinze segundos).

Tabela B.15: Número de Alternativas Mortas – Primeiros Casos de Teste Incluídos.

Número	Entrada			Número de Alternativas Mortas
	A	B	C	
1	0	1	2	9
2	1	2	0	17
3	2	0	1	5
4	2	1	0	0
5	1	0	2	0
6	0	2	1	75
7	0	1	1	1
8	1	1	0	0
9	1	0	1	0
10	0	2	0	1
11	0	0	2	0
12	2	0	0	0
13	2	2	2	0

- Verificação da Medida de Cobertura.

Total de Alternativas: 115
Total de Alternativas Anômalas: 0
Total de Alternativas Vivas: 7
Total de Alternativas Equivalentes: 0
Total de Casos de Teste: 13
MEDIDA DE COBERTURA: 0.93913

Figura B.6: *Medida de Cobertura* – Primeiro Resultado para o Maior de Três Números.

Com o resultado obtido, ainda existem alternativas vivas, então o testador decidiu incluir casos de teste (Tabela B.16), continuando a execução da sessão de teste, ou seja, executando o programa em teste e as alternativas com os novos casos de teste.

Tabela B.16: Casos de Teste Incluídos e Número de Alternativas Mortas.

Número	Entrada			Número de Alternativas Mortas
	A	B	C	
14	-2	2	2	0
15	2	-2	2	0
16	2	2	-2	0
17	-2	-2	-2	0
18	-3	-4	-5	0

Após a execução dos casos de teste incluídos foi verificado que as alternativas permaneceram vivas (Tabela B.17). Então, o testador analisou-as, concluindo que as mesmas são equivalentes ao programa em teste. Para determinar as alternativas equivalentes na GPTesT, o testador informou os números das alternativas a ferramenta. O resultado final pode ser visto na Figura B.7.

Tabela B.17: Alternativas para Encontrar o Maior de Três Números.

Número	Alternativa Gerada Viva
9	if(MT(B,A)) {if(LT(C,B)) {if(MT(B,B)) {if(LT(B,A)) {if(LT(A,C)) {Z =B;}} else {Z =A;}} else {Z =B;}} else {if(LT(A,C)) {Z =C;} else {Z =A;}}} else {if(MT(C,A)) {Z =C;} else {Z =A;}}
90	if(LT(B,B)) {if(MT(B,B)) {if(MT(B,A)) {if(MT(C,C)) {if(MT(C,C)) {if(LT(B,B)) {if(LT(C,A)) {if(LT(A,A)) {if(LT(B,B)) {if(LT(C,A)) {if(LT(C,A)) {Z =A;}}} else {if(MT(A,A)) {if(LT(C,B)) {if(MT(C,C)) {if(MT(A,C)) {if(LT(B,C)) {Z =A;}}}}}}} else {if(LT(B,B)) {if(LT(B,B)) {if(MT(C,B)) {Z =B;}} else {Z =C;}} else {if(LT(A,C)) {Z =A;}}}}} else {Z =B;}} else {Z =B;}} else {Z =A;}} else {if(LT(B,B)) {if(LT(B,A)) {if(MT(C,B)) {if(LT(C,A)) {if(LT(C,A)) {Z =A;}}}} else {Z =C;}} else {if(MT(C,A)) {if(MT(B,C)) {if(LT(C,A)) {if(MT(C,B)) {if(MT(A,B)) {if(LT(A,C)) {if(MT(A,B)) {Z =B;} else {Z =A;}} else {if(MT(B,A)) {Z =B;}}} else {if(LT(C,C)) {if(LT(B,C)) {Z =A;}}}} else {Z =B;}} else {Z =C;}} else {if(MT(A,B)) {if(MT(C,B)) {if(MT(A,C)) {Z =A;} else {Z =C;}} else {Z =A;}} else {Z =B;}}}}
94	if(LT(C,B)) {if(LT(A,B)) {Z =B;} else {Z =A;}} else {if(LT(A,C)) {Z =C;} else {Z =A;}}
97	if(MT(C,B)) {if(MT(A,C)) {if(LT(B,C)) {if(LT(B,A)) {if(LT(B,A)) {Z =A;}}}} else {Z =C;}} else {if(LT(B,A)) {Z =A;} else {Z =B;}}
105	if(LT(A,B)) {if(LT(C,B)) {if(MT(C,C)) {if(LT(B,A)) {if(MT(A,C)) {Z =C;}} else {Z =C;}} else {Z =B;}} else {Z =C;}} else {if(LT(A,C)) {if(LT(C,C)) {if(MT(C,C)) {if(MT(B,A)) {if(LT(A,A)) {Z =B;}} else {if(MT(B,A)) {Z =A;}}} else {Z =B;}} else {Z =C;}} else {Z =A;}}
107	if(LT(A,C)) {if(MT(C,B)) {Z =C;} else {if(LT(B,A)) {Z =A;} else {Z =B;}}} else {if(LT(B,A)) {Z =A;} else {Z =B;}}
114	if(MT(B,C)) {if(LT(C,A)) {if(MT(A,B)) {Z =A;} else {Z =B;}} else {Z =B;}} else {if(LT(A,C)) {Z =C;} else {Z =A;}}

Total de Alternativas: 115
Total de Alternativas Anômalas: 0
Total de Alternativas Vivas: 0
Total de Alternativas Equivalentes: 7
Total de Casos de Teste: 18

MEDIDA DE COBERTURA: 1

Figura B.7: Medida de Cobertura - Resultado Final para o Maior de Três Números.

A Tabela B.18 apresenta os resultados obtidos com o programa maior de três números, de um modo geral, através das colunas: Primeiro, Segundo e Final que se relacionam ao número de vezes que o GPTesT foi executado com novos casos de teste ou com a identificação de alternativas equivalentes. Nas linhas são fornecidos: o número total de alternativas geradas pela Chameleon; de alternativas diferentes ou ativas, que são consideradas para a execução da ferramenta; de alternativas anômalas, de alternativas vivas, de alternativas equivalentes, de casos de teste usados nas execuções até o momento, o tempo de execução e os resultados da medida de cobertura em cada execução da ferramenta.

Na Tabela B.19 estão disponíveis os casos de teste usados e o número de alternativas mortas com cada um deles.

Tabela B.18: Resultados Programa Maior de Três Números.

Resultado	Primeiro	Segundo	Final
Nº Alternativas Geradas			170
Nº de Alternativas Diferentes			115
Nº Alternativas Anômalos	0	0	0
Nº Alternativas Vivas	7	7	0
Nº Alternativas Equivalentes	0	0	7
Nº Casos Teste	13	18	18
Casos Teste			Tabela 6.5
Tempo de Execução	00:02:15h	00:00:49h	-x-
MEDIDA DE COBERTURA	0.93913	0.93913	1

Tabela B.19: Casos de Teste para o Programa Maior de Três Números.

Número do Caso de Teste	Caso de Teste			Número de Alternativas Mortas
	X	Y	Z	
1	0	1	2	9
2	1	2	0	17
3	2	0	1	5
4	2	1	0	0
5	1	0	2	0
6	0	2	1	75
7	0	1	1	1
8	1	1	0	0
9	1	0	1	0
10	0	2	0	1
11	0	0	2	0
12	2	0	0	0
13	2	2	2	0
14	-2	2	2	0
15	2	-2	2	0
16	2	2	-2	0
17	-2	-2	-2	0
18	-3	-4	-5	0

Programa Mínimo Múltiplo Comum

➢ Primeiro passo: Executar a Chameleon, obtendo as alternativas para a execução da GPTesT.

Os parâmetros utilizados na execução da Chameleon foram:

- o casos de teste (Tabela B.20);
- o configuração inicial *I* (Figura B.8);
- o número de rodadas, 10;
- o número de gerações, 100.

Tabela B.20: Casos de Teste para o Programa MMC.

Entrada		Saída
X	Y	Z
1	2	2
4	2	4
16	2	16
5	6	30
5	8	40
2	25	50
3	5	15
4	3	12
9	2	18
11	5	55
2	0	0
3	0	0

```
[begin]
[parameters]
population size = 500
tournament size = 10
maximum depth for initial random programs = 15
maximum depth for programs created during the run = 30
crossover rate = 90
mutation rate = 0
elitist = N
threshold = 0.01
[compiler]
cl -nologo -G6 -MT -Fepop.exe
[result-producing branch]
terminal set = {X,Y}
function set = {RESTO,NE,MUL,DIV}
output variable = Z
[result-producing branch productions]
<code>   -> <def> <prog> <result>
<def>    -> float R = 1, A = X, B = Y;
<result> -> Z = DIV(MUL(A,B),<var>);
<prog>   -> if (<expc>) {<prog1>} else {<atr>}
<prog1>  -> do {<bloco>} while (<expc>);
<bloco>  -> <exp>
<bloco>  -> <bloco> <exp>
<exp>    -> <var> = <opm>(<var>,<var>);
<exp>    -> <var> = <var>;
<expc>   -> <opc>(<var>,<cte>)
<atr>    -> <var> = <cte>;
<opm>    -> RESTO
<op>     -> MUL
<op>     -> DIV
<opc>    -> NE
<var>    -> X
<var>    -> Y
<var>    -> R
<cte>    -> 0
[fitness cases]
source -> mmc.dat
[end]
```

Figura B.8: Configuração Inicial / para o Programa MMC .

Na Tabela B.21 podem ser visualizadas algumas alternativas encontradas pela Chameleon para o problema do cálculo do mmc.

Tabela B.21: Exemplo de Alternativas Geradas para o MMC.

Rodada	Alternativa
1	float R = 1, A = X, B = Y;if (NE(R,0)) {do {Y =RESTO(X,Y);R =RESTO(R,R);} while (NE(Y,0));} else {R =0;}Z = DIV(MUL(A,B),R);
2	float R = 1, A = X, B = Y;if (NE(R,0)) {do {Y =RESTO(X,Y);R =RESTO(R,R);} while (NE(Y,0));} else {R =0;}Z = DIV(MUL(A,B),R);
3	float R = 1, A = X, B = Y;if (NE(R,0)) {do {R =Y;X =RESTO(X,R);Y =RESTO(Y,R);} while (NE(X,0));} else {Y =0;}Z = DIV(MUL(A,B),R);
4	float R = 1, A = X, B = Y;if (NE(X,0)) {do {R =RESTO(R,R);R =Y;X =RESTO(X,Y);Y =X;X =R;} while (NE(Y,0));} else {X =0;}Z = DIV(MUL(A,B),R);
5	float R = 1, A = X, B = Y;if (NE(R,0)) {do {Y =RESTO(X,Y);R =RESTO(R,R);} while (NE(Y,0));} else {R =0;}Z = DIV(MUL(A,B),R);
6	float R = 1, A = X, B = Y;if (NE(R,0)) {do {R =RESTO(R,R);Y =RESTO(X,Y);} while (NE(Y,0));} else {R =0;}Z = DIV(MUL(A,B),R);
7	float R = 1, A = X, B = Y;if (NE(R,0)) {do {Y =RESTO(X,Y);R =RESTO(R,R);} while (NE(Y,0));} else {R =0;}Z = DIV(MUL(A,B),R);
8	float R = 1, A = X, B = Y;if (NE(R,0)) {do {R =X;X =RESTO(Y,X);Y =R;R =RESTO(X,R);} while (NE(R,0));} else {X =0;}Z = DIV(MUL(A,B),Y);
9	float R = 1, A = X, B = Y;if (NE(Y,0)) {do {R =Y;Y =RESTO(X,Y);X =R;} while (NE(Y,0));} else {X =0;}Z = DIV(MUL(A,B),R);
10	float R = 1, A = X, B = Y;if (NE(R,0)) {do {X =X;R =RESTO(Y,X);Y =RESTO(X,Y);X =Y;X =RESTO(Y,Y);} while (NE(Y,0));} else {R =0;}Z = DIV(MUL(A,B),R);

➢ Segundo passo: gerar os dados de teste para a execução do programa em teste e das alternativas na GPTesT (Tabela B.22).

Tabela B.22: Dados de Teste para a Gptest – Programa MMC.

Número	Entrada	
	X	Y
1	0	2
2	1	0
3	6	3
4	2	4
5	3	7
6	0	0

➢ Terceiro passo: Executar a GPTesT com os resultados obtidos com a Chameleon; configuração inicial *I* (Figura B.8); o programa em teste (Figura 5.3); e os dados de teste (Tabela B.22).

Os resultados gerados com a execução podem ser observados na seqüência.

- Verificação de programas gerados idênticos pela Chameleon.

  Foram geradas 525 alternativas com 44 identificadas como alternativas diferentes.

- Execução do programa em teste.

Tabela B.23: Resultados da Execução do Programa em Teste para o MMC.

Número	Entrada		Saída
	X	Y	Z
1	0	2	0
2	1	0	0
3	6	3	6
4	2	4	4
5	3	7	21
6	0	0	0

- Execução das alternativas.

  Na Tabela B.24 são apresentadas as quantidades de alternativas mortas com a execução de cada caso de teste. O tempo de execução das alternativas foi de 00:00:46h (quarenta e seis segundos).

Tabela B.24: Alternativas Mortas após sua Execução.

Número	Entrada		Número de Alternativas Mortas
	X	Y	
1	0	2	23
2	1	0	0
3	6	3	16
4	2	4	2
5	3	7	0
6	0	0	0

- Verificação da Medida de Cobertura.

O resultado da medida de cobertura (Figura B.9) não foi satisfatório, optando-se por incluir mais um caso de teste (Tabela B.26) antes da identificação dos equivalentes. As alternativas vivas podem ser visualizadas na Tabela B.25. Com a execução do programa e das alternativas com o caso de teste número 7, as alternativas que permaneciam vivas, morreram e o resultado final da Medida de Cobertura foi 1, sem a necessidade de identificação de programas equivalentes.

```
Total de Alternativas: 44
Total de Alternativas Anômalas: 0
Total de Alternativas Vivas: 3
Total de Alternativas Equivalentes: 0
Total de Casos de Teste: 6
MEDIDA DE COBERTURA: 0.931818
```

Figura B.9: Medida de Cobertura para o MMC.

Tabela B.25: Possível Alternativa para o Cálculo do MMC.

Número	Alternativa Gerada Viva
15	float R = 1, A = X, B = Y;if (NE(X,0)) {do {R =RESTO(R,R);R =Y;X =RESTO(X,Y);Y =X;X =R;} while (NE(Y,0));} else {X =0;}Z = DIV(MUL(A,B),R);
34	float R = 1, A = X, B = Y;if (NE(R,0)) {do {R =X;X =RESTO(Y,X);Y =R;R =RESTO(X,R);} while (NE(R,0));} else {X =0;}Z = DIV(MUL(A,B),Y);
36	float R = 1, A = X, B = Y;if (NE(Y,0)) {do {R =Y;Y =RESTO(X,Y);X =R;} while (NE(Y,0));} else {X =0;}Z = DIV(MUL(A,B),R);

Tabela B.26: Caso de Teste Incluído para o MMC.

Número	Entrada	
	X	Y
7	-1	4

A Tabela B.27 apresenta os resultados obtidos com o programa mínimo múltiplo comum, de um modo geral, através das colunas: Primeiro e Final que se relacionam ao número de vezes que o GPTesT foi executado com novos casos de teste ou com a identificação de alternativas equivalentes. Nas linhas são fornecidos: o número total de alternativas geradas pela Chameleon; de alternativas diferentes ou ativas, que são consideradas para a execução da ferramenta; de alternativas anômalas, de alternativas vivas, de alternativas equivalentes, de casos de teste usados nas execuções até o momento, o tempo de execução e os resultados da medida de cobertura em cada execução da ferramenta.

Tabela B.27: Resultados Programa MMC.

Resultado	Primeiro	Final
Nº Alternativas Geradas		525
Nº de Alternativas Diferentes		44
Nº Alternativas Anômalos	0	0
Nº Alternativas Vivas	3	0
Nº Alternativas Equivalentes	0	0
Nº Casos Teste	6	7
Casos Teste		Tabela 6.7
Tempo de Execução	00:00:46h	00:00:08h
MEDIDA DE COBERTURA	0.931818	1

Na Tabela B.28 estão disponíveis os casos de teste usados e o número de alternativas mortas com cada um deles.

Tabela B.28: Casos de Teste para o Programa MMC.

Número do Caso de Teste	Caso de Teste		Número de Alternativas Mortas
	X	Y	
1	0	2	23
2	1	0	0
3	6	3	16
4	2	4	2
5	3	7	0
6	0	0	0
7	-1	4	3

Programa Máximo Divisor Comum

- Primeiro passo: Executar a Chameleon, obtendo as alternativas para a execução da GPTesT.

Os parâmetros utilizados na execução da Chameleon foram:

- casos de teste (Tabela B.29);
- configuração inicial *I* (Figura B.10);
- número de rodadas, 10;
- número de gerações, 100.

Tabela B.29: Casos de Teste para o Programa MDC.

Entrada		Saída
X	Y	Z
1	1	1
2	1	1
5	6	1
8	16	8
6	3	3
25	5	5
4	20	4
8	2	2
2	8	2
10	15	5
1	0	0
2	0	0
3	0	0
0	10	0
0	0	0

```
[begin]
[parameters]
population size = 500
tournament size = 7
maximum depth for initial random programs = 30
maximum depth for programs created during the run = 60
crossover rate = 90
mutation rate = 0
elitist = N
threshold = 0.01
[compiler]
cl -nologo -G6 -MT -Fepop.exe
[result-producing branch]
terminal set = {X,Y}
function set = {MOD,NE}
output variable = Z
[result-producing branch productions]
<code>  -> <def> <prog> <result>
<def>   -> float R=1;
<result> -> Z = <var>;
<prog>  -> do {<bloco>} while (<expc>);
<prog>  -> if (<expc>) {<prog>} else {<atr>}
<bloco> -> <exp>
<bloco> -> <bloco> <exp>
<exp>   -> <var> = <op>(<var>,<var>);
<exp>   -> <var> = <var>;
<expc>  -> <opc>(<var>,<cte>)
<expc>  -> <opc>(<var>,<cte>)
<atr>   -> <var> = <cte>;
<op>    -> MOD
<opc>   -> NE
<var>   -> Y
<var>   -> X
<var>   -> R
<cte>   -> 0
[fitness cases]
source -> mdc.dat
[end]
```

Figura B.10: Configuração Inicial I para o Programa MDC.

Como exemplo de soluções encontradas para o cálculo do mdc pela ferramenta Chameleon, com os parâmetros vistos acima, são apresentadas as alternativas da Tabela B.30.

Tabela B.30: Exemplo de Alternativas Geradas para o MDC.

Rodada	Alternativa
1	float R=1;do {R =X;X =MOD(Y,R);} while (NE(X,0));Z =R;
2	float R=1;if (NE(Y,0)) {if (NE(X,0)) {do {Y =MOD(Y,X);R =MOD(Y,R);R =MOD(Y,X);Y =X;X =MOD(R,Y);} while (NE(X,0));} else {Y =0;}} else {Y =0;}Z =Y;
3	float R=1;if (NE(Y,0)) {if (NE(X,0)) {do {R =X;Y =Y;X =MOD(Y,X);} while (NE(X,0));} else {R =0;}} else {R =0;}Z =R;
4	float R=1;do {Y =MOD(Y,X);X =MOD(X,X);} while (NE(X,0));Z =Y;
5	float R=1;if (NE(Y,0)) {if (NE(X,0)) {do {R =MOD(Y,R);R =MOD(R,R);X =MOD(Y,X);R =MOD(Y,X);} while (NE(R,0));} else {X =0;}} else {X =0;}Z =X;
6	float R=1;if (NE(X,0)) {if (NE(X,0)) {if (NE(X,0)) {if (NE(Y,0)) {do {X =MOD(Y,X);Y =MOD(X,X);} while (NE(Y,0));} else {X =0;}} else {X =0;}} else {X =0;}} else {X =0;}Z =X;
7	float R=1;if (NE(X,0)) {if (NE(Y,0)) {do {R =X;Y =Y;X =Y;X =MOD(X,R);Y =R;R =MOD(X,Y);} while (NE(R,0));} else {Y =0;}} else {Y =0;}Z =Y;
8	float R=1;if (NE(Y,0)) {if (NE(X,0)) {do {R =X;X =MOD(Y,X);} while (NE(X,0));} else {R =0;}} else {R =0;}Z =R;
9	float R=1;if (NE(Y,0)) {if (NE(Y,0)) {if (NE(X,0)) {do {R =Y;Y =X;X =MOD(X,R);X =MOD(R,Y);} while (NE(X,0));} else {Y =0;}} else {R =0;}} else {R =0;}Z =Y;
10	float R=1;if (NE(X,0)) {if (NE(Y,0)) {do {R =X;X =MOD(Y,R);X =Y;Y =X;X =MOD(Y,R);Y =R;R =Y;} while (NE(X,0));} else {R =0;}} else {R =0;}Z =R;

- Segundo passo: Gerar os dados de teste para o mdc ser executado na GPTesT. Os dados de teste selecionados podem ser visualizados na Tabela B.31.

Tabela B.31: Dados de Teste para a Gptest

Número	Entrada	
	X	Y
1	1	4
2	0	0
3	0	1
4	4	8
5	2	6
6	9	5
7	3	3
8	5	9

- Terceiro passo: Executar a GPTesT com os resultados obtidos com a Chameleon; configuração inicial *I* (Figura B.29); o programa em teste (Figura A.2); e os dados de teste (Tabela B.31).

Os resultados advindos da execução podem ser observados a seguir.

- Verificação de programas gerados idênticos pela Chameleon.

  Foram geradas 464 alternativas com 152 identificadas como alternativas diferentes.

- Execução do programa em teste.

  Na Tabela B.32, são apresentados os resultados obtidos com a execução do programa em teste com os casos de teste selecionados pelo testador.

Tabela B.32: Resultados da Execução do Programa em Teste para o Problema MDC.

Número	Entrada		Saida
	X	Y	Z
1	1	4	1
2	0	0	0
3	0	1	0
4	4	8	4
5	2	6	2
6	9	5	1
7	3	3	3
8	5	9	1

- Execução das alternativas geradas pela Chameleon.

  O tempo de execução das alternativas com os casos de teste mencionados acima foi de 00:01:28h (um minuto e vinte e oito segundos).

Tabela B.33: Número de Alternativas Mortas com a Execução da Gptest para o Caso MDC.

Número	Entrada		Número de Alternativas Mortas
	X	Y	
1	1	4	3
2	0	0	10
3	0	1	17
4	4	8	111
5	2	6	0
6	9	5	4
7	3	3	3
8	5	9	0

- Verificação da Medida de Cobertura.

  Na Figura B.11 é apresentado o primeiro resultado obtido com a execução da GPTesT no caso mdc.

Total de Alternativas: 152
Total de Alternativas Anômalas: 0
Total de Alternativas Vivas: 4
Total de Alternativas Equivalentes: 0
Total de Casos de Teste: 8

MEDIDA DE COBERTURA: 0.973684

Figura B.11: Medida de Cobertura para o MDC – Primeiro Resultado.

Na Tabela B.34 podem ser vistas alternativas vivas após a execução dos casos de teste já citados.

Tabela B.34: Alternativas Possíveis para o Cálculo do MDC.

Número	Alternativa Gerada Viva
13	float R=1;if (NE(Y,0)) {if (NE(X,0)) {do {Y =MOD(Y,X);R =MOD(Y,R);R =MOD(Y,X);Y =X;X =MOD(R,Y);} while (NE(X,0));} else {Y =0;}} else {Y =0;}Z =Y;
136	float R=1;if (NE(X,0)) {if (NE(Y,0)) {do {R =X;Y =Y;X =Y;X =MOD(X,R);Y =R;R =MOD(X,Y);} while (NE(R,0));} else {Y =0;}} else {Y =0;}Z =Y;
148	float R=1;if (NE(Y,0)) {if (NE(Y,0)) {if (NE(X,0)) {do {R =Y;Y =X;X =MOD(X,R);X =MOD(R,Y);} while (NE(X,0));} else {Y =0;}} else {R =0;}} else {R =0;}Z =Y;
152	float R=1;if (NE(X,0)) {if (NE(Y,0)) {do {R =X;X =MOD(Y,R);X =Y;Y =X;X =MOD(Y,R);Y =R;R =Y;} while (NE(X,0));} else {R =0;}} else {R =0;}Z =R;

Na tentativa de matar mais alternativas ainda vivas, foi incluído outro caso de teste (Tabela B.35), cujo resultado final gerado está apresentado na Figura B.12.

Tabela B.35: Caso de Teste Incluído para o MDC.

Número	Entrada	
	X	Y
9	-2	2

```
Total de Alternativas: 152
Total de Alternativas Anômalas: 0
Total de Alternativas Vivas: 0
Total de Alternativas Equivalentes: 0
Total de Casos de Teste: 9
MEDIDA DE COBERTURA: 1
```

Figura B.12: Medida de Cobertura para o MDC – Resultado Final.

A Tabela B.36 apresenta os resultados obtidos com o programa máximo divisor comum, de um modo geral, através das colunas: Primeiro e Final que se relacionam ao número de vezes que o GPTesT foi executado com novos casos de

teste ou com a identificação de alternativas equivalentes. Nas linhas são fornecidos: o número total de alternativas geradas pela Chameleon; de alternativas diferentes ou ativas, que são consideradas para a execução da ferramenta; de alternativas anômalas, de alternativas vivas, de alternativas equivalentes, de casos de teste usados nas execuções até o momento, o tempo de execução e os resultados da medida de cobertura em cada execução da ferramenta.

A Tabela B.37 apresenta os casos de teste usados e o número de alternativas mortas com cada um deles.

Tabela B.36: Resultados Programa MDC.

Resultado	Primeiro	Final
Nº Alternativas Geradas		464
Nº de Alternativas Diferentes		152
Nº Alternativas Anômalos	0	0
Nº Alternativas Vivas	4	0
Nº Alternativas Equivalentes	0	0
Nº Casos Teste	8	9
Casos Teste		Tabela 6.9
Tempo de Execução	00:01:28h	00:00:10h
MEDIDA DE COBERTURA	0.973684	1

Tabela B.37: Casos de Teste para o MDC.

Número do Caso de Teste	Caso de Teste		Número de Alternativas Mortas
	X	Y	
1	1	4	3
2	0	0	10
3	0	1	17
4	4	8	111
5	2	6	0
6	9	5	4
7	3	3	3
8	5	9	0
9	-2	2	4

Apêndice C – Resultados da execução dos programas com a GPTesT e a Proteum

As Tabelas C.1, C.3, C.5, C.7, apresentam os resultados obtidos com os programas fatorial, maior de três números, mínimo múltiplo comum e máximo divisor comum executados com as ferramentas GPTesT e Proteum. As colunas das Tabelas apresentam os resultados com as duas ferramentas em relação ao número de alternativas diferentes (para GPTesT) ou ativas (para Proteum); de alternativas anômalas; de alternativas equivalentes; de casos de teste usados nas execuções.

Tabela C.1: Resultados do Programa Fatorial Executado com Gptest e Proteum.

Resultado	GPTesT	Proteum
Nº Alternativas Ativas	402	263
Nº Alternativas Anômalos	0	0
Nº Alternativas Equivalentes	0	30
Nº Casos Teste	9	7

As Tabelas C.2, C.4, C.6, C.8, apresentam os casos de teste usados para as duas ferramentas nos programas já mencionados.

Tabela C.2: Casos de Teste para o Programa Fatorial em Gptest e Proteum.

Número do Caso de Teste	Caso de Teste	
	GPTesT	Proteum
	X	X
1	1	0
2	0	5
3	2	1
4	4	3
5	3	-1
6	5	vazio
7	6	2 3
8	7	- x -
9	-1	- x -

Tabela C.3: Resultados do Programa Maior de Três Números Executado com Gptest e Proteum.

Resultado	GPTesT	Proteum
Nº Alternativas Ativas	115	527
Nº Alternativas Anômalas	0	0
Nº Alternativas Equivalentes	7	85
Nº Casos Teste	18	35

Tabela C.4: Casos de Teste para o Programa Maior de Três Números em Gptest e Proteum.

Número do Caso de Teste	Caso de Teste					
	GPTesT			Proteum		
	a	b	c	a	b	C
1	0	1	2	1	2	3
2	1	2	0	1	3	2
3	2	0	1	2	1	3
4	2	1	0	2	3	1
5	1	0	2	3	1	2
6	0	2	1	3	2	1
7	0	1	1	5	5	5
8	1	1	0	0	0	0
9	1	0	1	-1	-2	-3
10	0	2	0	-1	2	3
11	0	0	2	1	2	-3
12	2	0	0	0	0	3
13	2	2	2	3	0	0
14	-2	2	2	0	3	0
15	2	-2	2	0	1	
16	2	2	-2	-1	-1	-1
17	-2	-2	-2	5	1	3
18	-3	-4	-5	1	5	3
19	- x -	- x -	- x -	5	6	3
20	- x -	- x -	- x -	-2	-1	-3
21	- x -	- x -	- x -	-2	-3	-1
22	- x -	- x -	- x -	-1	-3	-2
23	- x -	- x -	- x -	5	6	8
24	- x -	- x -	- x -	-1	1	-2
25	- x -	- x -	- x -	-4	-2	-3
26	- x -	- x -	- x -	-2	-4	-3
27	- x -	- x -	- x -	-4	-3	-2
28	- x -	- x -	- x -	1	6	6
29	- x -	- x -	- x -	6	6	1
30	- x -	- x -	- x -	6	1	6
31	- x -	- x -	- x -	1	2	3
32	- x -	- x -	- x -	0	0	-1
33	- x -	- x -	- x -	0	-1	0
34	- x -	- x -	- x -	-1	0	0
35	- x -	- x -	- x -	2	4	1

Tabela C.5: Resultados do Programa Mmc Executado com Gptest e Proteum.

Resultado	GPTesT	Proteum
Nº Alternativas Ativas	44	554
Nº Alternativas Anômalas	0	1
Nº Alternativas Equivalentes	0	39
Nº Casos Teste	7	17

Tabela C.6: Casos de Teste para o Programa Mmc em Gptest e Proteum.

Número do Caso de Teste	Caso de Teste				
	GPTesT		Proteum		
	a	b	a	B	
1	0	2	1	2	
2	1	0	4	2	
3	6	3	16	2	
4	2	4	5	6	
5	3	7	5	8	
6	0	0	2	25	
7	-1	4	3	5	
8	- x -	- x -	1	0	
9	- x -	- x -	0	1	
10	- x -	- x -	0	0	
11	- x -	- x -	-1	-1	
12	- x -	- x -	-1	2	
13	- x -	- x -	2	-1	
14	- x -	- x -	2		
15	- x -	- x -	2	1	4
16	- x -	- x -	2	1	
17	- x -	- x -	6	7	

Tabela C.7: Resultados do Programa Mdc Executado com Gptest e Proteum.

Resultado	GPTesT	Proteum
Nº Alternativas Ativas	152	397
Nº Alternativas Anômalas	0	1
Nº Alternativas Equivalentes	0	36
Nº Casos Teste	9	15

Tabela C.8: Casos de Teste para o Programa Mdc em Gptest e Proteum.

Número do Caso de Teste	Caso de Teste			
	GPTesT		Proteum	
	a	b	a	b
1	1	4	1	1
2	0	0	2	1
3	0	1	5	6
4	4	8	8	16
5	2	6	6	3
6	9	5	8	2
7	3	3	1	0
8	5	9	0	2
9	-2	2	0	0
10	- x -	- x -	-1	-1
11	- x -	- x -	1	-1
12	- x -	- x -	-1	1
13	- x -	- x -	1	
14	- x -	- x -	1	2 3
15	- x -	- x -	vázio	