INALI WISNIEWSKI SOARES

Análise de Mutantes e Critérios Restritos no Contexto de Teste de Software: Resultados de uma Avaliação Empírica

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
2000

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 ***Inali Wisniewski Soares***, avaliamos o trabalho intitulado **"Analise de Mutantes e Critérios Restritos no Contexto de Teste de Software: Resultados de Uma Avaliação Empírica"**, cuja defesa foi realizada no dia 06 de outubro de 2000. Após a avaliação, decidimos pela aprovação da Candidata.

Curitiba, 06 de outubro de 2000.

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

Prof. Dr. Mário Jino
DCA-FEEC/UNICAMP

Profª. Dra. Aurora Trinidad Ramirez Pozo
DINF/UFPR

Ao Marlon, à Maria Iraci e ao Vitoldo,
pelo incentivo e apoio dedicados.

Agradecimentos

- Agradeço inicialmente e especialmente à minha orientadora Profª. Drª. Silvia Regina Vergilio pelo apoio, esforço, dedicação, orientação e amizade.
- Ao Marlon e a toda minha família pelo apoio, incentivo, compreensão, amor e carinho.
- Às amigas Josiane e Luciane pela amizade sincera, companheirismo, apoio e incentivo.
- Aos professores e amigos do Departamento de Informática da Universidade Estadual do Centro Oeste - UNICENTRO pela compreensão e apoio dedicados.
- Aos professores do Departamento de Informática da Universidade Federal do Paraná.
- A todos os amigos conquistados na UFPR, especialmente ao Aldri, Cristiani, Leonardo, Rudá, Andrey, Patricia, Denis, Ivo e Alaine.
- Ao Prof. José Carlos Maldonado (USP – São Carlos) e à W. Eric Wong (Telecordia Technologies) por fornecerem os programas e casos de teste utilizados no experimento.

Resumo

A utilização dos produtos de software em praticamente todas as áreas da atividade humana tem gerado uma crescente demanda por qualidade e produtividade. Nesse sentido, a atividade de teste se tornou fundamental. Vários critérios de teste, baseados em diferentes princípios, têm sido propostos nos últimos anos com o objetivo de auxiliar na geração de conjuntos de dados de teste assim como na avaliação da adequação desses conjuntos. Entre esses, destacam-se os critérios estruturais, o critério baseado em erros, Análise de Mutantes e os Critérios Restritos que utilizam restrições para derivar os testes. Estudos teóricos e empíricos comparando esses critérios têm sido realizados com o objetivo de se obter uma estratégia de aplicação de baixo custo e alta eficácia. Eles são geralmente realizados considerando os fatores: custo, em termos do número de casos de teste necessários; eficácia, em termos de número de erros revelados e *strength* (dificuldade de satisfação). Os Critérios Estruturais e os Critérios Restritos são incomparáveis com o Critério Análise de Mutantes do ponto de vista teórico e apenas estudos empíricos podem mostrar a relação entre esses critérios. Este trabalho apresenta os resultados de uma avaliação empírica para comparação do critério Análise de Mutantes e o Critério Restrito, Todos-Potenciais-Usos Restritos, considerando os fatores acima mencionados. Esses resultados apontam uma relação empírica que é utilizada para propor uma estratégia de aplicação de diferentes critérios de teste. Para a aplicação do Critério Análise de Mutantes – um estudo sobre mutantes equivalentes, um problema da atividade de teste, foi realizado. Além dos resultados teóricos desse estudo, também são apresentados resultados empíricos sobre mutantes equivalentes. Esses resultados podem auxiliar na implementação de mecanismos para reduzir o efeito desse problema durante o teste.

Abstract

The use of software products in most areas of human activities has generated a growing interest in software quality assurance. In this sense, the testing activity is fundamental. Several testing criteria, based on different principles, have been proposed during recent years aiming at test data set generation as well at the adequacy analysis of these sets. Structural criteria, Mutation Analysis - an error based criterion, and Constraint Based Criteria are examples of testing criteria. Theoretical and empirical studies comparing these criteria have been accomplished with the goal of obtaining an economical and efficient strategy. They are generally conducted considering the factors: cost – number of test cases, efficacy – number of revealed errors and strength. The Mutation Analysis and Constraint Based criteria are theoretically incomparable; only empirical studies can point out the relationship between these criteria. This work presents the results from an empirical evaluation aimed at comparing the Mutation Analysis and All-Constrained-Potential-Uses criteria considering the above mentioned factors. The obtained results show an empirical relationship between the criteria, which is used as a basis of strategy proposed for application of the different testing criteria. During Mutation Analysis criterion application, a study about equivalent mutants, a problem for the testing activity, was performed. Theoretical and empirical results with respect to equivalent mutants, obtained from this study and from the empirical evaluation, are also presented. These results can used to support the implementation of mechanisms to reduce the effect of this testing problem.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contexto

Nas últimas décadas a Engenharia de Software evoluiu significativamente estabelecendo técnicas, critérios, métodos e ferramentas para produção de software. A utilização dos produtos de software em praticamente todas as áreas da atividade humana tem gerado uma crescente demanda por qualidade e produtividade.

O desenvolvimento de software está sujeito a vários tipos de erros. Na Engenharia de Software, atividades de teste e validação são fundamentais para garantir a qualidade de software. A escassez de tempo e de recursos humanos capacitados, agravados pela indisponibilidade de ferramentas adequadas, são os principais problemas enfrentados por uma equipe de teste.

O teste de software envolve as seguintes etapas: planejamento de testes, projeto, execução e avaliação dos casos de testes. Essas atividades devem ser realizadas durante o processo de desenvolvimento de software e geralmente, são aplicadas nas seguintes fases: 1) teste de unidade - identificação de defeitos em cada módulo do software; 2) teste de integração - teste baseado no projeto e na arquitetura do software; 3) teste de validação - avaliar satisfação de requisitos estabelecidos na fase de análise; 4) teste de sistema - teste do software com outros elementos do sistema.

Diversas técnicas de teste têm sido propostas para selecionar bons casos de teste e assim revelar a maioria dos defeitos utilizando um mínimo de tempo e esforço. Geralmente as técnicas utilizadas são: Técnica Funcional - deriva os casos de teste baseados na função de especificação do software; Técnica Estrutural - baseia-se na estrutura interna do programa para derivar os casos de teste; Técnica Baseada em Erros - os casos de teste são obtidos através do conhecimento dos principais erros que geralmente ocorrem durante o processo de desenvolvimento do software.

Essas técnicas devem ser vistas como complementares e as melhores vantagens de cada uma devem ser exploradas produzindo dessa forma testes eficazes e de baixo custo. Elas estabelecem critérios de teste, que podem ser utilizados tanto para auxiliar na geração de conjuntos de dados de teste como para auxiliar na avaliação da adequação desses conjuntos.

Entre os vários critérios propostos para se conduzir e avaliar a qualidade da atividade de teste, destacam-se os critérios estruturais, baseados em fluxo de controle e baseados em fluxo de dados [Rap82, Nta84, UY88, Mal91], e o critério Análise de Mutantes [Dem78], baseado em erros.

Os critérios estruturais utilizam uma representação de programa conhecida como grafo de fluxo de controle para determinar as estruturas que devem ser cobertas pelos casos de teste. Um grafo de fluxo de controle é um grafo orientado onde cada vértice representa um bloco indivisível de comandos e cada aresta representa um desvio de um bloco para outro.

Os critérios baseados em fluxo de dados associam informações sobre o uso das variáveis ao grafo de fluxo de controle, determinando associações entre pontos onde um valor é atribuído a uma variável (chamados de definição de variável) e pontos onde esse valor é utilizado (chamados de uso da variável).

Basicamente, a idéia do critério Análise de Mutantes [Dem78] é criar a confiança de que um programa P está correto produzindo-se, através de pequenas alterações sintáticas, um conjunto de programas, chamados de mutantes, semelhantes a P, e construindo-se dados de teste capazes de detectar diferenças de comportamento entre P e seus mutantes.

Uma ferramenta automatizada é de fundamental importância para a aplicação dos critérios de teste; sem isso, apenas programas muito simples podem ser testados. Existem várias ferramentas de teste, entre essas podem ser citadas como exemplo: as ferramentas de teste Proteum [Del93] e Mothra [Dem88] que apóiam a utilização do critério Análise de Mutantes, e as ferramentas Poke-Tool [Cha91], Atac [Hor92] e Asset [Fra85] que apóiam a utilização dos critérios Baseados em Fluxo de Dados.

No entanto, existem limitações, inerentes à própria atividade de teste, para a completa automatização e utilização dos critérios de testes; por exemplo, a existência de caminhos não executáveis e de mutantes equivalentes.

Um caminho é não executável se não existe dado de teste que o execute. Um mutante é equivalente a um programa P se possui comportamento idêntico a P para todas as entradas.

Determinar caminhos não executáveis [Fra87] e mutantes equivalentes [Bud81] são questões indecidíveis. Alguns estudos têm sido realizados relativos aos problemas de caminhos não executáveis [HH85, Fra87, MYV90, Ver92] e mutantes equivalentes [BS79, Bud81, OC94, OJ97] e visam a propor soluções para minimizar seus efeitos na aplicação e automatização dos critérios de teste.

Gerar conjuntos de dados de teste é o processo de identificar dados de entrada para o programa que satisfaçam um critério selecionado. São encontradas diversas categorias de técnicas utilizadas para gerar dados de teste para os critérios existentes. Uma delas é a geração de dados de teste sensíveis a erros, que possui elevada probabilidade de revelar alguns tipos de erros específicos. Estão incluídas nessa categoria as técnicas: Teste de Domínio [WC80]; Teste Baseado em Restrições [DMO91]; Teste de Operador Relacional e Booleano e Teste Operador Relacional e Booleano com parâmetro $\in$ [Tai93].

Visando a aumentar a eficácia dos critérios estruturais, Vergilio [Ver97] introduziu os "Critérios Restritos". Esses critérios permitem a utilização de critérios estruturais juntamente com os princípios de técnicas de geração de dados de testes sensíveis a erros, introduzindo uma restrição que descreve um particular

tipo de erro. Essa restrição deverá ser satisfeita pelo dado de teste gerado, para satisfazer um dado critério estrutural.

Os critérios de teste mencionados acima são complementares e devem ser utilizados em conjunto a fim de se obter uma atividade de teste de boa qualidade. Por isso vários estudos teóricos e empíricos têm sido realizados com o objetivo de comparar os diversos critérios de teste.

Os critérios estruturais baseados em fluxo de controle e baseados em fluxo de dados têm sido comparados [FW88, Mal91] utilizando-se principalmente a complexidade – número de casos de teste necessários para satisfazer o critério no pior caso para qualquer programa P e a relação de inclusão – estudo da ordem parcial entre os critérios.

Um critério C_1 inclui um critério C_2 se todo e qualquer conjunto de caminhos que satisfaz C_1, ou é C_1-adequado também satisfaz C_2, ou é C_2-adequado. Se C_1 não inclui C_2 e C_2 não inclui C_1 diz-se que C_1 e C_2 são incomparáveis .

A relação de inclusão estabelece certas propriedades dos critérios estruturais e requisitos mínimos necessários que devem ser satisfeitos, por exemplo, incluir o critério Todos-Ramos e do ponto de vista de fluxo de dados, ao menos um uso de todo resultado computacional [Mal91]. Os critérios estruturais e o critério Análise de Mutantes são incomparáveis; por isso, a maioria das comparações são feitas empiricamente.

A condução de estudos empíricos [FW88, Wey90, Mal91, Wey93, Mat93, Mat94, Wong94a, Sou96, Ver97] tem sido intensificada nos últimos anos procurando, através da comparação entre os critérios existentes, definir uma estratégia confiável e econômica para a realização da atividade de teste, em que o custo, a eficácia e a dificuldade de satisfação (*strength*) são fatores básicos para comparar a adequação de um critério de teste.

Custo refere-se ao esforço necessário para a utilização de um critério; a eficácia refere-se à capacidade de um critério de revelar a presença de um maior número de defeitos em relação a outro; e a dificuldade de satisfação está associada à probabilidade de satisfazer um critério tendo satisfeito outro.

O critério baseado em erros, Análise de Mutantes tem se mostrado mais eficaz em termos de número de defeitos revelados do que os critérios estruturais; porém, é mais custoso em termos de número de casos de teste [WMM94, Sou96].

Vergilio [Ver97], utilizando uma extensão da ferramenta Poke-Tool [Cha91] realizou um experimento de validação dos Critérios Restritos, comparando o critério Todos-Potenciais-Usos (um critério estrutural baseado em fluxo de dados) com o critério Todos-Potenciais-Usos Restritos. Resultados mostraram um aumento da eficácia dos dados gerados, aplicando-se os Critérios Restritos. Entretanto, ainda não existe nenhum estudo realizado para comparar o critério Análise de Mutantes e os Critérios Restritos.

1.2 Motivação

A partir da apresentação do contexto onde este trabalho se insere, pode-se destacar os principais motivos para o seu desenvolvimento:

- O teste de software é uma das atividades mais importantes entre as atividades de verificação e validação de produtos de software;
- A necessidade de métodos, técnicas e critérios de teste que permitam aumentar a qualidade e reduzir os custos do teste;
- Estudos empíricos são fundamentais para avaliar o custo e a eficácia em revelar erros, e permitem demonstrar características importantes e aspectos complementares entre as técnicas de teste;
- A necessidade de estratégias de testes que apresentem baixo custo e alta eficácia em revelar a presença de erros;
- O esforço gasto para determinar mutantes equivalentes e caminhos não executáveis que são grandes limitações para a aplicação dos critérios de teste;
- O critério Análise de Mutantes é mais efetivo porém mais custoso porque exige muitas execuções do programa e em geral necessita um número maior de casos de teste em estudos empíricos;

- Os Critérios Restritos mostraram-se mais eficazes que o critério estrutural [Ver97], mas como são incomparáveis com o critério Análise de Mutantes, somente estudos empíricos poderão mostrar a relação entre esses critérios.

1.3 Objetivos

O principal objetivo desse trabalho é realizar uma avaliação empírica para comparação dos critérios Restritos e Análise de Mutantes. Os resultados da avaliação empírica permitem:

- Comparar os critérios com relação ao custo em termos do número de casos de teste necessários e eficácia, em termos de número de erros revelados e *strength* (dificuldade de satisfação);
- Verificar as principais características dos mutantes equivalentes gerados, com o objetivo de contribuir para reduzir o efeito desse problema;
- Estabelecer uma estratégia de teste para aplicação dos diferentes critérios de teste existentes.

1.4 Organização

Esse capítulo apresentou o contexto no qual esse trabalho está inserido, a motivação para realizá-lo e o objetivo a ser atingido.

No Capítulo 2 são introduzidos a terminologia e os principais conceitos relacionados à atividade de teste e ao trabalho realizado.

O Capítulo 3 apresenta os resultados do estudo empírico realizado comparando os Critérios Restritos e o critério Análise de Mutantes em termos de custo, eficácia e *strength*.

No Capítulo 4 são apresentados os resultados relacionados ao experimento de aplicação do critério Análise de Mutantes descrito no Capítulo 3, referentes aos mutantes equivalentes encontrados durante o estudo empírico. Também é

apresentado um resumo de técnicas que podem ser utilizadas para automatizar a determinação de mutantes equivalentes e facilitar a aplicação do critério Análise de Mutantes.

O Capítulo 5 apresenta as conclusões, contribuições e perspectivas de trabalhos futuros.

O trabalho contém dois apêndices. No Apêndice A são apresentados os operadores de mutação disponíveis na ferramenta Proteum. No Apêndice B são apresentados os dados coletados para um dos programas do experimento.

Capítulo 2

Revisão Bibliográfica

Nesse capítulo são apresentados conceitos básicos envolvendo a atividade de teste, bem como a terminologia e critérios de teste correspondente. São feitas considerações sobre a importância do teste de software durante o processo de desenvolvimento; em seguida são apresentadas as principais técnicas e critérios de teste, destacando o teste estrutural e o teste baseado em erros. Uma visão geral é apresentada sobre técnicas de geração de dados de teste, com ênfase na geração de dados de testes sensíveis a erros.

Os critérios Restritos e o critério Análise de Mutantes são discutidos com mais detalhes devido à importância deles nesse trabalho. O desenvolvimento de ferramentas que automatizem a atividade de teste é bastante relevante para que os testes possam ser conduzidos de forma satisfatória e livre de erros.

Finalmente é apresentada uma síntese dos principais estudos empíricos conduzidos utilizando-se os critérios Baseados em Fluxo de Dados e o critério Análise de Mutantes, bem como os resultados obtidos.

2.1 Definição de Erro, Defeito e Falha

A IEEE tem realizado vários esforços de padronização, incluindo a terminologia usada em teoria de teste. O padrão IEEE número 610.12-1990 [I390] é utilizado para definir os seguintes termos: **defeito** (*fault*) – problema no código

fonte do programa; **engano** (*mistake*) - ação humana que produz um resultado incorreto; **erro** *(error)*: diferença entre o valor obtido e o valor esperado, e **falha** (*failure*) - produção de uma saída incorreta com relação à especificação.

Segundo Howden [How76], os erros/defeitos de programa se classificam em: erros de domínio, erros de computação e erros de estrutura de dados. Na literatura existem diferentes classificações para erros/defeitos [GG75, How76, BP84, OW84, DMM95]. Abaixo está a classificação que será utilizada ao longo desse trabalho.

Erros de Domínio:

O domínio de um caminho é formado por todas as entradas que executam tal caminho, sendo que a cada domínio corresponde uma função. Um erro de domínio ocorre se um domínio incorreto for associado a um caminho. Existem dois tipos de erros de domínio: erro na seleção de caminho - existe um erro em um predicado em alguma parte do caminho e um caminho incorreto é executado; erro de caminho ausente – não existe um caminho no programa correspondente a uma parte do domínio de entrada ou especificação do programa.

Erros de Computação:

O erro provoca uma computação incorreta mas o caminho executado é igual ao caminho esperado. Erros de computação se dividem em dois tipos: a computação existe no programa mas existe um defeito; e a computação está ausente no programa.

Erros em Estrutura de Dados:

Um erro em estruturas de dados corresponde à definição incorreta de uma estrutura de dados.

2.2 Teste de Software

Nas últimas décadas a Engenharia de Software evoluiu significativamente estabelecendo técnicas, critérios, métodos e ferramentas para produzir software. A utilização dos produtos de software em praticamente todas as áreas da

atividade humana gera uma crescente demanda por qualidade e produtividade. Por isso, a etapa de teste é de grande importância para a identificação e eliminação de defeitos que persistem, representando assim o último passo da especificação, projeto e codificação [Pre92].

Normalmente, o teste é realizado em quatro etapas: teste de unidade, teste de integração, teste de validação e teste de sistema. O teste de unidade, procura identificar defeitos de lógica e de implementação em cada unidade do software. O teste de integração é uma técnica sistemática para integrar os módulos componentes da estrutura do software, visando a identificar defeitos de interface entre os módulos. O teste de validação valida os requisitos de software estabelecidos na fase de análise. O teste de sistema é realizado após a integração do sistema e visa a identificar defeitos de funções e características de desempenho, que não estejam de acordo com a especificação.

A realização do teste é composta basicamente das seguintes atividades: construção de casos de teste, definindo o dado de entrada e especificando a saída esperada; execução do programa com os dados gerados e análise do comportamento do programa para verificar se existem defeitos. O processo é repetido até que o testador tenha confiança no programa.

Segundo Myers [Mye79], teste é o processo de executar um programa com a intenção de encontrar um defeito; um bom caso de teste é aquele com alta probabilidade de revelar a presença de defeito e um teste bem sucedido é aquele que detecta a presença de um defeito ainda não descoberto.

Em princípio, o programa deveria ser executado com todas as combinações possíveis do domínio de entrada, mas o teste exaustivo é impraticável devido às restrições de tempo e de custo. Por isso, várias técnicas e critérios de teste têm sido definidos para projetar casos de teste com o objetivo de revelar a maioria dos defeitos existentes, respeitando as restrições de tempo e custo associados ao desenvolvimento de um software.

2.3 Técnicas de Teste de Software

Os critérios de teste de software são estabelecidos a partir de três técnicas básicas: a funcional, a estrutural e a baseada em erros. Essas técnicas diferenciam-se pela origem da informação utilizada na avaliação e construção dos conjuntos de casos de teste, sendo que a cada uma está associado um conjunto de critérios para esse fim [Mal91].

Deve-se notar que nenhuma das técnicas de teste é completa, no sentido de que nenhuma delas é, em geral, suficiente para garantir a qualidade da atividade de teste. Na verdade, essas diferentes técnicas se complementam e devem ser aplicadas em conjunto para assegurar um teste de boa qualidade [Pre92].

2.3.1 Técnica Funcional

Essa técnica também é conhecida como teste de caixa preta pois trata o software como uma caixa cujo conteúdo é desconhecido e só é possível ver o lado externo, ou seja, os dados de entrada fornecidos e as respostas produzidas como saída. Nessa técnica são verificadas as funções do sistema sem se preocupar com detalhes de implementação.

Os critérios de teste funcional mais conhecidos são [Pre92]:

- **particionamento em classes de equivalência:** é um critério de teste funcional que divide o domínio de entrada de um programa em classes, a partir das quais os casos de teste podem ser derivados. O objetivo é minimizar o número de casos de teste, selecionando-se somente um caso de teste de cada classe;
- **análise do valor limite:** esse critério é um complemento ao critério particionamento em classes de equivalência, que procura testar os limites de cada classe de equivalência;
- **grafo de causa e efeito**: utiliza-se esse critério para testar o efeito combinado de dados de entrada. Causas e efeitos são identificados e combinados em um grafo a partir do qual uma tabela de decisão é criada.

Os dados de teste e as saídas são derivados da tabela de decisão obtida.

2.3.2 Técnica Estrutural

A técnica estrutural, conhecida como teste de caixa branca, leva em consideração os aspectos de implementação na escolha dos casos de teste. Normalmente, requer que determinados elementos do programa sejam exercitados por caminhos do grafo de fluxo de controle associado ao programa, dessa forma um critério estrutural é estabelecido. Esses critérios são referenciados como critérios baseados em caminhos.

Um grafo de fluxo de controle é um grafo orientado, com um único nó de entrada e um único nó de saída, no qual cada vértice representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para outro. Um exemplo de grafo de fluxo de controle é apresentado na Figura 2.1. Um caminho é uma seqüência finita de nós ($n_1, n_2, ..., n_k$), $k \geq 2$, tal que existe um arco de n_i para n_{i+1} para i = 1,2,...,k-1. Um caminho completo é um caminho onde o primeiro nó é o nó de entrada e o último nó é o nó de saída do grafo G.

```
void soma ( )
{
   float soma;
   int i,n;
   1    soma = 1;
   1    scanf("%d",&n);
   1,2,3  for (i=2;i<=n;i++)
   3          soma = soma +pow(i,2);
   4    printf ("%f",soma);
}
```

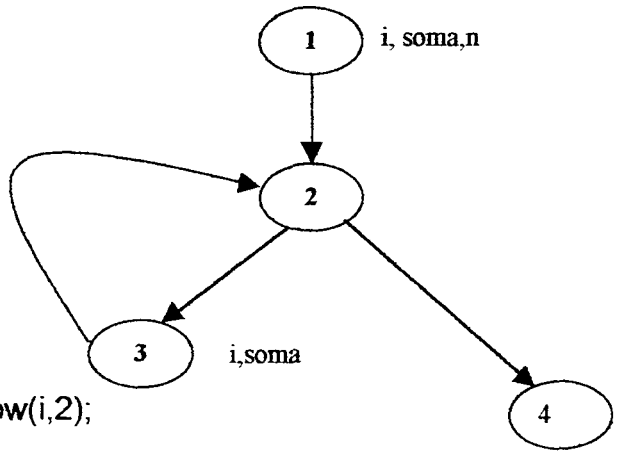


Figura 2.1: Exemplo de um programa e o seu grafo de fluxo de controle

Os critérios estruturais baseiam-se em tipos de estruturas diferentes para determinar quais partes do programa são requeridas na execução, os critérios estruturais mais conhecidos e utilizados são apresentados nas próximas seções.

2.3.2.1 Critérios Baseados em Complexidade

Utilizam informações sobre a complexidade do programa para obter os requisitos de teste. Um critério bastante conhecido desta classe é o critério de McCabe que utiliza a complexidade ciclomática para derivar os requisitos de teste, descrito a seguir:

- **Critério Todos Caminhos Linearmente Independentes** - [McC76, Pre92]: exige que um conjunto de caminhos independentes do programa seja executado. Um caminho independente deve incluir no mínimo um arco do grafo de fluxo de controle que não faça parte de qualquer outro caminho do conjunto. O número de caminhos em um conjunto de caminhos independentes do programa é dado pela complexidade *ciclomática do programa.*

2.3.2.2 Critérios Baseados em Fluxo de Controle

Nestes critérios, características de controle da execução do programa, tais como comandos ou desvios, são utilizadas para determinar quais estruturas são necessárias; os critérios mais conhecidos são:

- **Critério Todos-Nós**: exige que cada nó do grafo seja exercitado pelo menos uma vez.
- **Critério Todos-Ramos:** exige que cada arco do grafo seja exercitado pelo menos uma vez.
- **Critério Todos-Caminhos:** exige ao menos uma execução de cada caminho (seqüência finita de nós) do grafo.

2.3.2.3 Critérios Baseados em Fluxo de Dados

Os requisitos de teste desses critérios são obtidos através das informações do fluxo de dados do programa. Uma característica comum é que eles requerem a execução de caminhos a partir da definição da variável até onde ela foi utilizada.

Para derivar os requisitos de teste requeridos por esses critérios é necessário adicionar ao grafo de fluxo de controle informações sobre o fluxo de dados, como o Grafo Def-Uso (Def-Use Graph) definido por Rapps e Weyuker [Rap82, Rap85]. Neste grafo são exploradas as associações entre a definição e o uso das variáveis determinando os caminhos a serem exercitados.

Uma definição de variável ocorre quando um valor é armazenado em uma posição de memória. Um uso de uma variável ocorre quando é feita uma referência a essa variável. O uso da variável pode ser computacional (c-uso), quando é usada em uma computação, e predicativo (p-uso), quando afeta diretamente o fluxo de controle do programa. Com base nessas associações, são determinados os requisitos de teste. Um caminho $(i,n_1,...,n_m, j)$ é livre de definição com relação a uma variável x (c.r.a x), do nó i para o nó j ou arco (n_m,j), se nenhuma definição de x ocorre nos nós n_1 a n_m. Um caminho é simples se todos os nós exceto, possivelmente, o primeiro e o último são distintos; é livre de laços se todos os nós são distintos.

Os principais critérios de Rapps e Weyuker [RW85] são:

- **Critério Todos-Usos**: exige que todas as associações do tipo (i,j,x) ou (i,(j,k),x), sejam cobertas, onde, o nó i possui uma definição de x e o nó j possui um uso de x ou (respectivamente o arco (j,k), possui um uso de x em um predicado), e ainda, existe um caminho livre de definição c.r.a x de i para j (ou para o arco (j,k)).
- **Critério Todos-Du-Caminhos:** exige que todos os du-caminhos do programa sejam cobertos. Um du-caminho c.r.a x é dado por uma seqüência de nós $(n_1,...,n_j,n_k)$, onde n_1 possui uma definição de x e n_k possui um uso de x e $(n_1,...,n_j,n_k)$ é um caminho livre de definição c.r.a x (ou ainda, (n_j,n_k) possui um uso de x em um predicado e $(n_1,...,n_j,n_k)$ é um caminho livre de definição c.r.a x e $(n_1,...,n_j)$ é um caminho livre de

laços).

A partir dos critérios de Rapps e Weyuker, Maldonado [Mal91] definiu os critérios Potenciais-Usos, introduzindo o conceito de potencial-associação. Esses critérios não exigem o uso explícito de uma variável para que uma associação seja estabelecida; ou seja, requerem que caminhos livres de definição a partir da definição de uma variável sejam executados, independentemente de ocorrer uso dessa variável nesse caminho. Esses critérios são:

- **Critério Todos-Potenciais-Usos**: exige que todas as associações potenciais-usos, do tipo (i,j,x) ou (i,(j,k),x) sejam cobertas, onde existe uma definição de x no nó i e existe pelo menos um caminho livre de definição c.r.a x do nó i para o nó j (ou arco(j,k)).
- **Critério Todos-Potenciais-Du-Caminhos:** exige que todos os potenciais-du-caminhos do programa sejam cobertos. Um potencial-du-caminho c.r.a x é dado por uma seqüência de nós $(n_1,...,n_j,n_k)$, onde n_1 possui uma definição de x e $(n_1,...,n_j,n_k)$ é livre de definição c.r.a x do nó n_1 para n_j (ou arco(n_j,n_k)), e $(n_1,...,n_j)$ é livre de laços.
- **Critério Todos-Potenciais-Usos/Du**: exige que todas as potenciais associações sejam cobertas por potenciais-du-caminhos.

Um dos problemas relativos aos critérios da técnica estrutural é a impossibilidade de determinar automaticamente se um caminho é ou não executável; ou seja, não existe algoritmo tal que, dado um caminho completo qualquer, forneça o conjunto de valores que causam a execução desse caminho; não existe algoritmo nem mesmo para decidir se esse conjunto existe [Fra87]. Dessa forma, uma dificuldade existente é a necessidade de se determinar manualmente os caminhos não executáveis para o programa em teste.

2.3.3 Técnica Baseada em Erros

Utiliza informações sobre os tipos de erros mais freqüentes no processo de desenvolvimento de software para derivar os requisitos de teste. A ênfase da técnica está nos erros que o programador ou projetista pode cometer durante o

desenvolvimento e nas abordagens que podem ser usadas para detectar sua ocorrência. A Análise de Mutantes e a Semeadura de Erros são critérios típicos que se concentram em erros [Bud81].

Ao utilizar o critério Semeadura de Erros é necessário inserir alguns defeitos no programa; então é verificado durante o teste quais dos defeitos revelados foram inseridos (semeados) no programa e quais são naturais. A idéia é verificar a relação entre o número de defeitos semeados revelados e o número de defeitos naturais revelados e obter uma indicação do número de defeitos naturais restantes [Bud81].

A Análise de Mutantes é um critério que utiliza um conjunto de programas ligeiramente modificados, denominados mutantes, obtidos a partir de um programa P, para avaliar quanto um conjunto de casos de teste T é adequado para o teste de P. O objetivo é encontrar um conjunto de casos de teste T capaz de revelar as diferenças de comportamento existentes entre P e seus mutantes [Dem87].

2.3.3.1 Análise de Mutantes

O critério Análise de Mutantes surgiu na década de 70 na *Yale University* e *Georgia Institute of Technology*, possuindo um forte relacionamento com um método clássico para detecção de erros lógicos em circuitos digitais, conhecido como modelo de teste de falha única [Fri75].

A idéia básica do critério Análise de Mutantes foi apresentada por DeMillo em 1978 [Dem78], conhecida como hipótese do programador competente, assume que programadores experientes escrevem programas corretos ou muito próximos do correto. De acordo com essa hipótese, os defeitos nos programas são introduzidos através de pequenos desvios sintáticos e produzem um resultado incorreto. O objetivo é revelar esses defeitos, então programas mutantes são gerados a partir do programa em teste. Casos de teste devem ser construídos para diferenciar o comportamento do programa original do comportamento de um programa mutante. O mutante é morto quando ocorre essa diferença. O critério Análise de Mutantes exige que todos os mutantes sejam

mortos.

Uma outra hipótese explorada na aplicação do critério Análise de Mutantes é o efeito de acoplamento [Dem78], o qual assume que defeitos complexos estão relacionados a erros simples. Assim sendo, espera-se que conjuntos de casos de teste capazes de revelar defeitos simples sejam também capazes de revelar defeitos complexos.

Durante a Análise de Mutantes um programa P é testado com um conjunto de casos T cuja adequação deseja-se avaliar. O programa é executado com T e, se apresentar resultados incorretos, então um erro foi revelado e o teste termina. Caso contrário, o programa ainda pode conter defeitos que o conjunto T não foi capaz de revelar. O programa P sofre alterações, gerando os programas P_1, P_2 ... P_n que são mutantes de P, os mutantes são gerados a partir de um conjunto de operadores que são específicos para uma linguagem de programação cujo objetivo é inserir mutações baseando-se nos defeitos mais comumente cometidos.

O passo seguinte é a execução dos mutantes com o mesmo conjunto de casos de teste T. Caso o comportamento de P_i (mutante de P) seja diferente de P, então esse mutante é considerado morto. Caso contrário, existem duas razões para esse mutante estar vivo:

1) o conjunto de casos de teste T não é suficiente para distinguir o comportamento de P e P_i e, novos casos de teste devem ser incluídos ao conjunto; ou

2) P_i é considerado equivalente a P, ou seja não existe um caso de teste capaz de matar P_i.

Um conjunto de casos de teste é adequado a um programa em teste, depois que todos os mutantes vivos foram mortos ou determinados como equivalentes [Dem78].

O escore de mutação é definido para determinar o grau de adequação do conjunto de casos de teste utilizado. O cálculo é efetuado da seguinte forma:

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

onde : DM(P,T) é o número de mutantes mortos pelo conjunto de casos de teste T.

P M(P) é o número de mutantes gerados para o programa P.

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

O escore de mutação varia no intervalo [0,1], e quanto maior o escore, mais adequado é o conjunto de casos de teste para o programa sendo testado. Através dessa fórmula é possível verificar que apenas DM(P,T) depende do conjunto de casos de teste utilizado enquanto que EM(P) é obtido à medida que os mutantes equivalentes são detectados.

Pode-se encontrar a descrição de como aplicar a Análise de Mutantes em vários trabalhos [Dem78, Acr80, Del93]. Basicamente, dado um programa P e um conjunto de casos de teste T, os seguintes passos são seguidos:

1) Geração dos mutantes

Os mutantes são gerados através da aplicação de operadores de mutação no programa P sendo testado. Um operador de mutação é obtido através de regras que definem as alterações que devem ser aplicadas no programa original P. Na maioria das vezes, a aplicação de um operador de mutação gera mais que um mutante, visto que P pode conter várias entidades que estão no domínio de um operador e, desse modo, o operador é aplicado a cada uma dessas entidades, uma de cada vez [Del93].

A escolha de um conjunto de operadores depende dos defeitos que se quer revelar, da cobertura que se quer garantir, da linguagem de programação em que os programas a serem testados foram escritos e da ferramenta de teste utilizada. Um exemplo são os 22 operadores de mutação para o teste de programas da linguagem Fortran definidos na ferramenta Mothra [Dem88, Cho89], outro exemplo são os 71 operadores de mutação existentes na ferramenta Proteum [Del93] para o teste de programas da linguagem C, esses operadores encontram-se no Apêndice A.

2) Execução do Programa

Esse passo na Análise de Mutantes é igual ao realizado em outros critérios de teste. Deve-se executar o programa original P usando os casos de teste selecionados e verificar se o resultado é o esperado. Caso o programa apresente um resultado incorreto para algum caso de teste, então um defeito foi revelado e o processo termina. A tarefa de oráculo, ou seja, decidir se o resultado está correto ou não, geralmente é desempenhada pelo testador. Isso não ocorre apenas no critério Análise de Mutantes, mas na maioria das técnicas e critérios de teste [Mal91].

3) Execução dos Programas Mutantes

Esse passo pode ser totalmente automatizado. Cada um dos mutantes gerados é executado usando-se os casos de teste fornecidos pelo testador. Os resultados são comparados com o resultado obtido através da execução do programa original. O próprio sistema de mutação marca como morto os mutantes que apresentarem resultados diferentes do resultado do programa original.

4) Análise dos mutantes vivos

A análise dos Mutantes vivos é o passo que requer mais intervenção humana. Inicialmente, os mutantes vivos são analisados para decidir se tais mutantes são equivalentes ou não ao programa original. Se o mutante é equivalente, então ele deve ser descartado. Caso contrário o conjunto de casos de teste não foi capaz de diferenciá-lo do programa original, e é necessário a inclusão de novos casos de teste ao conjunto. Deve-se então retornar ao segundo passo do critério e seguir os demais até que se consiga um bom conjunto de casos de teste T.

Deve-se observar que o procedimento para aplicar o critério Análise de Mutantes exige que algumas tarefas sejam executadas automaticamente, devido à grande quantidade de processamento de que elas necessitam. Tarefas como geração e execução de mutantes e comparação dos resultados obtidos demandam grande esforço de processamento, sendo praticamente impossível executá-las de modo eficiente e preciso sem o auxílio de uma ferramenta. Assim, faz-se necessário o uso de ferramentas que automatizem a aplicação do critério

[Del93].

A aplicação do critério Análise de Mutantes é exemplificado através da Figura 2.2 que apresenta a rotina *main* do programa *Cal,* o símbolo ■ representa duas possíveis mutações:

- A primeira mutação representada por *if (argc <1)* é equivalente ao programa original;
- Já a segunda mutação representada por *if (m>1 || m>12)* não é equivalente ao programa original, isto é, deve existir algum caso de teste capaz de mostrar a diferença de comportamento entre esse mutante e o programa original.

```
main(int argc, char *argv[])
{
        register y, i, j;
        int m;
        if (argc == 2)
           goto xlong;
        if(argc < 2) {
  ■     if(argc < 1) {
           time_t t;
           struct tm *tm;
           t = time(0);
           tm = localtime(&t);
           m = tm->tm_mon + 1;
           y = tm->tm_year + 1900;
         } else {
              m = atoi(argv[1]);
              if (m<1 || m>12) {
           ■  if (m>1 || m>12) {
                 fprintf(stderr, "cal: %s: Bad month.\n", argv[1]);
                 exit(1);
              }
              y = atoi(argv[2]);
              if(y<1 || y>9999) {
                   printf(stderr, "cal: %s: Bad year.\n", argv[2]);
                   exit(2);
              }
        }
        printf("   %s %u\n", smon[m-1], y);
        printf("%s\n", dayw);
        cal(m, y, string, 24);
        for(i=0; i<6*24; i+=24)
           pstr(string+i, 24);
        exit(0);}
```

Figura 2.2: Rotina *main* do programa *Cal* em teste e dois de seus possíveis mutantes.

2.4 Geração de Dados de Teste

A tarefa de geração de dados de teste é o processo de identificar um conjunto de dados de entrada que satisfaçam o critério de teste selecionado. Um dos principais objetivos dentro da atividade de software é tornar essa tarefa automatizada independentemente do critério de teste selecionado; porém, existem limitações associadas à atividade de teste que dificultam a realização desse objetivo.

2.4.1 Principais Limitações Associadas à Geração de Dados de Teste

- **Correção Coincidente**: Existe um defeito no programa, mas mesmo após a execução do caminho que contém esse defeito, um resultado correto coincidentemente é produzido pelo programa;
- **Caminho Ausente**: Caso não exista um caminho no programa que implemente uma determinada função, não é garantido que a técnica estrutural exigirá o teste dessa função e o erro pode não ser revelado;
- **Caminhos Não Executáveis:** Quando não existe um conjunto de valores que possa ser atribuído às variáveis de entrada, variáveis globais e parâmetros do programa, que causa a execução desse caminho, diz-se que ele é não executável [Fra87];
- **Mutantes Equivalentes:** O problema de mutantes equivalentes para o critério Análise de Mutantes é semelhante ao problema de caminhos não executáveis para os critérios baseados em caminhos. Um mutante equivalente é aquele que se comporta igual ao programa original, para todos os dados do domínio do programa, ou seja, não existe um dado de teste que os diferencie. A determinação de mutantes equivalentes é uma questão indecidível [DMO91].

2.4.2 Técnicas de Geração de Dados de Testes

Técnicas de geração de dados de teste buscam selecionar bons dados de teste, e dessa forma satisfazer os critérios de testes. Existem diferentes abordagens para gerar dados de teste: geração aleatória de dados [DN84, HT88]; geração com execução simbólica [BEK75, Cla76, RSBC76, How77]; geração com execução dinâmica [Kor90]; e geração de dados sensíveis a erros: Teste de Domínio [WC80]; Teste Baseado em Restrições [DMO91]; Teste de Operador Relacional e Booleano e Teste Operador Relacional e Booleano com parâmetro $\in$ [Tai93].

A técnica de geração de dados sensíveis a erros visa a selecionar pontos do domínio relacionados com certos tipos de erros, que possuem uma alta probabilidade de revelar esses erros. Essas técnicas são de particular interesse neste trabalho e serviram como motivação para a introdução dos Critérios Restritos [Ver97], e são descritas a seguir:

2.4.2.1 Teste de Domínios

Teste de Domínio (Domain Testing-DT) é uma técnica de geração de dados de teste proposta por White e Cohen [WC80]. A estratégia é baseada em uma análise geométrica dos limites do domínio, cujo objetivo é selecionar os dados de teste no limite do domínio do caminho ou muito próximo a ele, e assim encontrar erros de domínio. Os limites do domínio são obtidos através das condições que são associadas aos ramos ao longo do caminho. A cada condição corresponde um predicado que é uma combinação lógica de expressões relacionais.

Os predicados dos programas são assumidos como predicados simples segundo White e Cohen [WC80]. Interpretações de predicados em condições de caminhos indicam os limites do domínio do caminho dados por uma borda, onde cada borda é a divisão do limite determinado por apenas um predicado simples na condição do caminho. As bordas podem ser abertas - os predicados possuem um dos operadores relacionais $>$, $<$, $\neq$; ou fechadas - os operadores relacionais associados aos predicados são $\leq, \geq, =$.

Os autores definiram alguns princípios para simplificar a estratégia construída: os programas utilizados não possuem *arrays*, ponteiros nem chamadas de funções e/ou procedimentos; correção coincidente não ocorre; erro de caminho ausente não é associado ao caminho em teste; cada borda refere-se a um predicado simples; o caminho correspondente a cada domínio adjacente computa uma função diferente do caminho em teste; a borda dada é linear, e se estiver incorreta, a borda correta é também linear e o espaço de entrada é contínuo.

Os pontos de teste selecionados são de dois tipos, definidos através de suas posições em relação à borda dada. Um ponto de teste *on* encontra-se na borda referida e um ponto de teste *off* encontra-se a uma distância ϵ, muito pequena e deve estar no domínio que não contém a borda. Ao testar uma borda fechada, os pontos *on* estão no domínio do caminho sendo testado e todo ponto *off* está em algum domínio adjacente. Por outro lado, ao testar uma borda aberta, todos os pontos *on* estão em algum domínio adjacente, enquanto os pontos *off* estão no domínio sendo testado.

2.4.2.2 Teste Baseado em Restrições

Teste Baseado em Restrições (Constraint-Based Testing - CBT) é uma técnica para auxiliar a geração de dados de teste adequados ao teste de mutação e foi proposta por DeMillo e Offut em 1991 [DMO91]. A técnica CBT utiliza restrições algébricas para descrever casos de teste e dessa forma encontrar tipos específicos de defeitos em programas.

Três condições são necessárias para que um caso de teste consiga matar um mutante, isto é, encontrar os defeitos inseridos pelos operadores de mutação nos programas mutantes: 1) **Condição de alcançabilidade**: é necessário alcançar o comando mudado; 2) **Condição de necessidade**: o estado do programa original e o estado do programa mutante devem ser diferentes, logo após a execução do comando mudado, esse estado intermediário é necessário mas não é suficiente para matar o mutante; 3) **Condição de suficiência**: o estado

incorreto deve ser propagado até o final do programa e uma falha deve ser produzida.

Derivar e encontrar casos de testes que satisfaçam a condição de suficiência é desejável mas é impraticável [DMO91]. O problema de suficiência tem sido considerada por outros pesquisadores e é semelhante ao problema da correção coincidente.

Para exemplificar as condições necessárias exigidas para matar um mutante é apresentado um programa na Figura 2.3, chamado MIN, que calcula o valor mínimo entre dois números M e N. Existem 4 comandos no programa que são enumerados, e os comandos mudados apresentam à sua esquerda a letra grega Δ. Para que o comando 1 seja alcançado não existe nenhuma restrição, já para o comando 3 ser alcançado o valor de N deve ser menor que M. As condições de necessidade para que estados diferentes sejam produzidos após a execução dos comandos 1 e 3 são $M \neq N$ e $N \neq 1$ nessa ordem.

```
        VOID MIN( )
{
        INT M,N,MIN;
1       MIN = M
Δ       MIN = N
2       IF (N < M)
3       MIN = N
Δ       MIN = 1
4       RETURN MIN
}
```

Figura 2.3 - Programa MIN: Condições necessárias para matar um mutante.

2.4.2.3 Teste Baseado em Predicados

Teste Baseado em Predicados é uma abordagem para testar software e requer certos tipos de testes para cada predicado do programa. Geralmente essas técnicas se dividem em dois grupos: teste de predicado simples e teste de predicado composto [Tai93]. A questão principal em testar predicados compostos

é o problema da propagação de defeito em um predicado composto, que refere-se à propagação de um resultado incorreto de qualquer parte de um predicado composto para um resultado incorreto na avaliação final do predicado. Pois se os predicados individuais que constituem o predicado composto forem avaliados individualmente e um defeito for encontrado, ainda assim o resultado da avaliação do predicado poderá ser coincidentemente correto. Essa situação pode ser exemplificada através da Figura 2.4.

Predicado P≠	**((E_1< E_2) ^**	**(E_3 ≥ E_4)) V**	**(E_5= E_6)**	**Resultado de P≠**
t_1	t	t	t	t
t_2	f	f	f	f

Figura 2.4 - Exemplo de avaliação de predicados

O conjunto de teste {t1,t2}não distingue P≠ do predicado ((E1< E2) V (E3 ≥ E4)) V (E5= E6), que difere de P≠ apenas nos operadores booleanos.

Tai [Tai93] propôs três técnicas de teste para predicados compostos. As técnicas propostas visam encontrar defeitos em predicados compostos, e requerem que dados de teste sejam executados para satisfazer um conjunto de restrições. Esse conjunto de restrições é construído para detectar defeitos em operadores booleanos, operadores relacionais e expressões aritméticas e garantem a avaliação incorreta do predicado.

Tai [Tai93] demonstra que a utilização de restrições BRO associadas ao critério Análise de mutantes em predicados que possuem operadores relacionais permitem aumentar a performance desse critério.

A seguir são descritas as três técnicas propostas por Tai:

- **Teste de Operador Booleano (ou BOR)** para um predicado composto visa garantir a detecção de um ou mais defeitos em operadores booleanos. Necessita-se executar os ramos true e false do predicado.
- **Teste de Operador Booleano e Relacional (ou BRO)** requer um conjunto de testes para um predicado composto para garantir a detecção de (um ou mais) defeitos em operadores relacionais e booleanos. Os

requisitos de teste de operador relacional pode ser definido como cobertura do conjunto de restrições {(>),(=),(<)}.

- **Teste de Expressão Relacional e Booleana (ou BRE) com parâmetros** ϵ , $\epsilon > 0$, requer um conjunto de teste para um predicado composto que garanta a detecção de (um ou mais) defeitos de operadores booleanos, defeitos de operadores relacionais e defeitos em expressões aritméticas. O conjunto de restrições {(+ ϵ),(=), (-ϵ)} é um caso especial do conjunto de restrições {(>), (=), (<)} e seu uso para detecção de expressões aritméticas incorretas foi sugerido por K. A. Foster [Fos80], um valor pequeno ϵ faz as restrições (+ϵ), (-ϵ) mais difíceis de serem cobertas, mas também faz essas restrições mais eficazes para detectar defeitos.

Tai [Tai93] ressalta que as técnicas BOR e BRO baseiam-se em defeitos em predicados para geração das restrições e conseqüentemente dos testes; o conjunto de restrições e teste gerados causariam a morte de todos os mutantes que descrevessem um defeito em operadores booleanos e relacionais, gerados por exemplo pelo operador ORRN da ferramenta Proteum, entre outros operadores, descritos no Apêndice A.

2.5 Critérios Restritos

Os Critérios Restritos introduzidos por Vergilio [Ver97], são extensões para as diversas famílias de critério de teste estrutural e visam a aumentar a eficácia desses critérios, ou seja, uma capacidade maior desses critérios revelarem erros, utilizando os fundamentos de técnicas de geração de dados sensíveis a erros. A cada elemento requerido é associado uma restrição a ser satisfeita durante a execução do caminho que exercitará esse elemento.

Mais de um caminho candidato a cobrir cada elemento requerido poderá ser exercitado, ou ainda um caminho poderá ser executado com um ou mais dados de teste que satisfarão as restrições que descrevem erros e portanto têm alta probabilidade de revelar um erro ainda não descoberto.

Critérios Restritos requerem elementos restritos, um elemento requerido é assim denominado porque seu domínio de entrada foi restringido. Quando uma restrição é associada a um elemento, o domínio de entrada do elemento requerido gerado fica limitado aos pontos que satisfazem a restrição dada.

Abaixo, são descritos os Critérios Restritos para os dois grupos de critérios estruturais mais conhecidos: critérios Baseados em Fluxo de Controle e critérios Baseados em Fluxo de Dados.

1. Critérios Restritos Baseados em Fluxo de Controle

- **Critério Todos-Nós Restritos**: a cada nó são associadas uma ou mais restrições. Um nó restrito é dado pelo par (i,C) e será coberto se um caminho que cobre o nó i for executado, e se C for satisfeita durante a sua execução.
- **Critério Todos-Ramos Restritos**: a cada ramo são associadas uma ou mais restrições. Um ramo restrito é dado pelo par ((i,j),C) e será coberto se um caminho que cobre o ramo (i,j) for executado, e se C for satisfeita durante a sua execução.
- **Critério Todos-Caminhos Restritos**: a cada caminho completo são associadas uma ou mais restrições. Um caminho restrito é dado pelo par (P,C), e será coberto se C for satisfeita durante a sua execução.

2. Critérios Restritos Baseados em Fluxo de Dados

- **Critério Todos-Usos Restritos**: a cada associação requerida são associadas uma ou mais restrições. Uma associação restrita é dada por ((i,j,x),C) ou (i,(j,k),x),C) e será coberta se um caminho livre de definição c.r.a x, ou seja um caminho que cobre a associação correspondente for executado, e se C for satisfeita durante a sua execução.
- **Critério Todos-Du-Caminhos Restritos**: associam-se a cada du-caminho uma ou mais restrições. Um du-caminho restrito é dado pelo par (DP,C) e será coberto se um caminho completo que inclui DP for executado, e se C for satisfeita durante a sua execução.

- **Critério Todos-Potenciais-Usos Restritos**: associam-se a cada potencial-associação requerida uma ou mais restrições. Uma potencial-associação restrita é dada por ((i,(j,k),x),C) e será coberta se um caminho livre de definição c.r.a x, ou seja um caminho que cobre a associação correspondente for executado, e se C for satisfeita durante a sua execução.

- **Critério Todos-Potenciais-Usos/Du Restritos**: associam-se a cada potencial-associação requerida uma ou mais restrições. Uma potencial associação restrita é dada por ((i,(j,k),x),C) e será coberta se um potencial-du-caminho livre de definição c.r.a x, ou seja um potencial-du-caminho que cobre a associação correspondente for executado, e se C for satisfeita durante a sua execução.

- **Critério Todos-Potenciais-Du-Caminhos Restritos**: associam-se a cada potencial-du-caminho requerido uma ou mais restrições. Um potencial-du-caminho restrito é dado pelo par (PDP,C) e será coberto se um caminho completo que inclui PDP for executado, e se C for satisfeita durante a sua execução.

Segundo Vergilio [Ver97], as restrições a serem associadas aos elementos requeridos por esses critérios estruturais, podem descrever qualquer tipo de erro. Para o estabelecimento dessas restrições sugere-se a utilização de princípios de diferentes técnicas de teste. As restrições associadas ao elemento devem ser restrições de necessidade, ou seja, devem descrever condições necessárias para revelar o erro.

Uma mesma restrição poderá estar associada a diferentes elementos requeridos e ser satisfeita ao mesmo tempo com mais de uma restrição de cobertura aumentando a probabilidade dessa conjunção de satisfazer a condição de suficiência, e do erro ser revelado.

Possíveis restrições foram derivadas, utilizando as técnicas de teste: Análise de Mutantes, BOR/BRO/BRE, Análise do Valor Limite e Teste de Computações.

As restrições são estruturadas em diferentes níveis. A cada nível, novas restrições vão sendo acrescentadas e uma quantidade maior de informação sobre

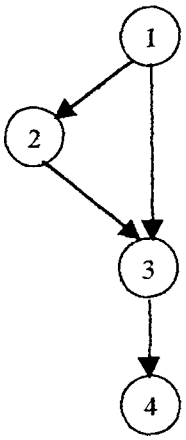
o programa será necessária. A seguir são apresentadas exemplos de possíveis restrições em cada nível:

- **Nível 1:** informação disponível: a associação na forma (i,(j,k),x) ou (i,j,k). O operador de mutação da ferramenta Proteum *VDTR - Domain Traps* (descrito no Apêndice A), e o critério de teste funcional, Análise do Valor Limite são aplicados às variáveis x das associações.
- **Nível 2:** informação disponível: um conjunto de associações. O conjunto poderá ser da forma $(i,(j,k),x_1)$, $(i,(j,k),x_2)$,..., $(i,(j,k),x_n)$ ou (i,j,x_1), (i,j,x_2),..., (i,j,x_n), de modo que a intersecção do conjunto de caminhos candidatos a cobrir as associações seja diferente do conjunto vazio. Restrições de necessidade associadas aos operadores de mutação de variáveis *Variable Mutation* (descritos no Apêndice A) são aplicadas às variáveis x_1... x_n. Pode-se também ter outras restrições além de $X \neq Y$, tais como $X < Y$, $X > Y$, $X = Y$, etc.
- **Nível 3:** informação disponível: o predicado no nó j, para uma associação do tipo (i,(j,k),x). Para o critério todos p-usos a existência de predicados é verdadeira. Quando não existir predicado, não serão geradas restrições. As técnicas BOR/BRO/BRE são aplicadas nesse nível. Restrições do conjunto T (que garantem a avaliação true do predicado) ou F (que garantem a avaliação "false" do predicado) são escolhidas de modo que o arco (j,k) seja executado.

Para exemplificar a utilização do critério Todos-Potenciais-Usos-Restritos, uma extensão do critério estrutural Todos-Potenciais-Usos, é apresentado um exemplo obtido de [Ver97]. O programa utilizado para o exemplo é o programa *Max* que calcula o máximo entre dois números, na Figura 2.5(a) é apresentado o código fonte, na Figura 2.5(b) o grafo de fluxo de controle, e para facilitar o entendimento também são apresentados na Figura 2.4(c) os elementos requeridos pelo critério Todos-Potenciais-Usos. As restrições possíveis em cada nível são descritas através da Tabela 2.1, e as associações restritas para o programa *Max* são descritas na Tabela 2.2.

```
   void max ( )
  { float m, max, n;
1   scanf("%f",&m);
1   scanf("%f",&n);
1   max = abs (m);
1   if (n>=m)
2      max = n;
3   printf("%f",max);
4   }
```

a) Código Fonte



b) Grafo de Fluxo de Controle

Elementos requeridos	Caminhos
(1,(1,3),{m,max,n}	1 3 4
(1,(1,2),{m,max,n}	1 2 3 4
(2,(-,-), {max}	

c) Elementos requeridos pelo critério Todos-Potenciais-Usos

Figura 2.5 - Programa *Max* utilizado para exemplificar a aplicação dos Critérios Restritos

Tabela 2.1 - Possíveis restrições em cada nível

Nível/Informação	Técnica Utilizada	Condição de Necessidade
1 (i,j,k,x)	Mutação de Comandos	$X = 0$, $X < 0$, $X > 0$ $A[e_1] > 0$, $A[e_1] < 0$, $A[e_1] = 0$ $S.c = 0$, $S.c > 0$, $S.c < 0$ p=null
2 um conjunto: (i,j,x_1) (i,j,x_2) .... (i,j,x_n)	Mutação de Variáveis	$A[e_1] \neq B[e_1]$ $p \neq q$ $A[e_1] \neq X$ $S.c \neq X$ $(*p) \neq X$, $Y \neq X$ $S \neq T$, $S.c \neq T.c$ $S.c_1 \neq S.c_2$
3 (i,(j,k),x) e predicado em (j,k)	BRO/BRE C_1 or C_2 C_1 and C_2	$F(C) = S_{1f}$ % S_{2f} $T(C) = \{ S_{2t} * \{f_2\}$ \$ $\{\{f_1\} * S_{2t}\}$ onde $f_1 \in S_{1f}$ e $f_2 \in S_{2f}$ e $(f_1, f_2) \in F(C)$ $T(C) = S_{1t}$ % S_{2t} $F(C) = \{ S_{1f} * \{t_2\}$ \$ $\{\{t_1\} * S_{2f}\}$ onde $t_1 \in S_{1t}$ e $t_2 \in S_{2t}$ e $(t_1, t_2) \in T(C)$

Tabela 2.2 - Associações restritas para o programa *Max* da Figura 2.5

Associação	Nível 1	Nível 2	Nível 3
(1,(1,2),n)	$n > 0$, $n < 0$, $n = 0$	$n \neq m$, $n \neq max$, $max \neq m$	$n > m$, $n = m$
(1,(1,2),m)	$m > 0$, $m < 0$, $m = 0$	$n \neq m$, $n \neq max$, $max \neq m$	$n > m$, $n = m$
(1,(1,2),max)	$max > 0$, $max < 0$, $max = 0$	$n \neq m$, $n \neq max$, $max \neq m$	$n > m$, $n = m$
(1,(1,3),n)	$n > 0$, $n < 0$, $n = 0$	$n \neq m$, $n \neq max$, $max \neq m$	$n > m$, $n = m$
(1,(1,3),m)	$m > 0$, $m < 0$, $m = 0$	$n \neq m$, $n \neq max$, $max \neq m$	$n > m$, $n = m$
(1,(1,3),max)	$max > 0$, $max < 0$, $max = 0$	$n \neq m$, $n \neq max$, $max \neq m$	$n > m$, $n = m$
(2,(-,-),max)	$max > 0$, $max < 0$, $max = 0$		

2.6 Ferramentas de Teste

A crescente complexidade do software, exige a utilização de ferramentas para auxiliar a atividade de teste. Sem a existência de uma ferramenta automatizada a aplicação de um critério torna-se uma atividade propensa a erros e utilizada somente para testar programas simples.

A disponibilidade de ferramentas automatizadas permite que a atividade de teste seja realizada em ambientes industriais produzindo dessa forma softwares de alta qualidade. Essas ferramentas também auxiliam pesquisadores a comparar, selecionar e estabelecer estratégias de teste.

A ferramenta de teste Poke-Tool [Cha91], desenvolvida no DCA/FEEC/Unicamp, foi implementada em linguagem C para possibilitar o uso dos critérios Potenciais-Usos. Ela permite a execução de um conjunto de casos de teste e faz a análise de adequação do conjunto executado em relação a esses critérios. Esta ferramenta é multi-linguagem, e pode ser configurada para diversas linguagens de programação tais como C, Pascal, FORTRAN e COBOL. Uma extensão à ferramenta Poke-Tool foi proposta em [Ver97] e implementada para dar apoio à utilização dos critérios Potenciais-Usos Restritos.

A Proteum (Program Testing Using Mutants) [Del93] desenvolvida no Instituto de Ciências Matemáticas e de Computação - ICMC-USP, é uma ferramenta de apoio ao critério Análise de Mutantes para programas escritos na linguagem C. Devido ao seu aspecto multi-linguagem ela pode ser configurada para o teste de programas escritos em outras linguagens de programação.

Os recursos oferecidos pela ferramenta são:

- definição de casos de teste;
- execução do programa em teste;
- seleção dos operadores de mutação que serão utilizados para geração dos mutantes;
- geração dos mutantes;
- execução dos mutantes com os casos de teste definidos;
- análise dos mutantes remanescentes e cálculo do escore de mutação

Na Proteum foram definidos 71 operadores de mutação divididos em quatro classes (Apêndice A): mutação de comandos (statement mutations), mutação de operadores (operators mutations), mutação de variáveis (variable mutations) e mutação de constantes (constant mutations). É possível escolher os operadores de acordo com as classes de erro que se deseja enfatizar, permitindo a geração de mutantes em etapas ou dividindo a atividade entre vários testadores. Depois de selecionar os operadores de mutação, são gerados os programas mutantes que representam as modificações efetuadas no programa original.

Diversas características adicionais foram incorporadas à Proteum para facilitar a condução de experimentos. É o caso, por exemplo da possibilidade de executar um mutante com todos os casos de teste disponíveis, para verificar quais casos de teste são mais eficientes.

O escore de mutação é obtido após os mutantes terem sido executados, e o testador deve decidir se o teste deve continuar ou não. Se o teste continuar, os mutantes vivos serão analisados para verificar sua equivalência e novos casos de teste serão necessários para matar os demais mutantes.

2.7 Avaliação e Comparação entre Critérios de Teste

Como existem muitos critérios de teste é difícil saber quais critérios utilizar e como utilizá-los de forma complementar para obter o melhor resultado com o menor custo.

Segundo Wong [Won93], custo, eficácia e dificuldade de satisfação (*strength*), são fatores básicos para comparar critérios de teste. **Custo**: é o esforço necessário para utilizar um critério. É medido através do número de casos de teste requeridos para satisfazer o critério ou por outras métricas dependentes do critério. **Eficácia**: é a capacidade de um critério em revelar uma quantidade maior de erros em relação a outro. **Dificuldade de satisfação (*strength*)**: procura verificar o quanto consegue-se satisfazer um critério C_1 tendo sido satisfeito um critério C_2.

Considerando esses fatores, estudos teóricos e empíricos são conduzidos para aumentar a produtividade e diminuir os custos da realização dos testes. Os estudos teóricos buscam estabelecer propriedades e características dos critérios de teste, *tal como a eficácia de uma estratégia de teste ou uma relação de* inclusão entre os critérios. Nos estudos empíricos são coletados dados e estatísticas que fornecem diretrizes para a escolha entre os diversos critérios disponíveis.

2.7.1 Comparação entre os Critérios Estruturais

Os critérios estruturais têm sido comparados através de uma relação de inclusão e de uma medida chamada complexidade do critério. A complexidade de um critério C é dada pelo número de casos de teste requeridos para satisfazer o critério no pior caso.

A relação de inclusão estabelece uma ordem parcial entre os diversos critérios. *Dado os critério* C_1 *e* C_2 *diz-se que* C_1 *inclui* C_2 *se todo e qualquer* conjunto de caminhos que satisfaz C_1, ou é C_1-adequado, também satisfaz C_2, ou também é C_2-adequado. O critério C_1 inclui estritamente C_2, descrito por $C_1 \Rightarrow C_2$, se C_1 inclui C_2 e C_2 não inclui C_1 [FW88]. Dado os critérios C_1 e C_2, diz-se que eles são incomparáveis se C_1 não incluir C_2 e C_2 não incluir C_1.

Estudos empíricos foram conduzidos para determinar o custo de aplicação dos critérios Baseados em Fluxo de Dados do ponto de vista prático. Weyuker [Wey90, Wey93] realizou um estudo para determinar um modelo para estimativa do número de casos de teste necessários para satisfazer um dado critério C, para um programa P. Os mesmos programas utilizados por Weyuker foram utilizados em estudos empíricos para avaliar os critérios Potenciais Usos [Mal91, Ver92]; esses estudos indicaram que os critérios baseados em fluxo de dados são aplicáveis.

Vergilio [Ver97] realizou um estudo onde comparou o critério Todos-Potenciais-Usos e o critério Todos-Potenciais-Usos-Restritos em termos de casos de teste necessários e eficácia desses critérios; esse estudo indicou a viabilidade de uso dos Critérios Restritos. O critério Todos-Potenciais-Usos-Restritos é mais

caro que o correspondente critério estrutural, porém mais eficaz que o critério Todos-Potenciais-Usos. Esse último fator justifica a aplicação dos Critérios Restritos em sistemas onde uma alta confiabilidade é requerida.

2.7.2 Estudos Empíricos relacionados ao Critério Análise de Mutantes

Estudos empíricos mostram que o critério Análise de Mutantes é bastante eficaz para detectar defeitos [Mat93, Mat94, Wong94a]. Esses estudos demonstram como a Análise de Mutantes se relaciona com os outros critérios de teste e buscam novas estratégias a fim de reduzir os custos associados aos critérios. Os critérios Baseados em Fluxo de Dados e o critério Análise de Mutantes são incomparáveis por isso, a maioria das comparações são feitas empiricamente.

Wong et al. [Won94a] utilizaram a Mutação Aleatória (10%) e a Mutação Restrita para comparar o critério Análise de Mutantes com o critério Todos-Usos. O objetivo era verificar o custo, eficácia e dificuldade de satisfação desses critérios. Nesse trabalho os autores concluíram que o critério Todos-Usos requer menos casos de teste que os dois critérios de mutação (Restrita e Aleatória nessa ordem). Quanto à eficácia a ordem referente ao critério mais eficaz para o menos é Mutação Restrita, Todos-Usos e Mutação Aleatória. Observa-se, com isso, que examinar apenas uma pequena porcentagem de mutantes pode ser uma abordagem útil para avaliar e construir conjuntos de casos de teste na prática. Portanto, a idéia é usar o critério Análise de Mutantes para testar partes críticas do software e utilizar alternativas mais econômicas como Mutação Restrita ou o critério Todos-Usos para testar as demais partes do software.

Um estudo empírico foi conduzido por Souza [Sou96] para avaliar o *strength* e o custo do critério Análise de Mutantes em relação aos critérios Potenciais-Usos [Mal91], os quais incluem o critério Todos-Usos. Os resultados indicaram que o custo do critério Análise de Mutantes foi maior que o custo dos critérios Potenciais-Usos. A dificuldade de satisfação (*strength*) obtida indica que os critérios Análise de Mutantes e Todos-Potenciais-Usos são incomparáveis mesmo

do ponto de vista empírico. Já os resultados obtidos nesse estudo indicam que os critérios Todos-Potenciais-Usos/Du e Todos-Potenciais-Du-Caminhos apresentam maior *strength* que o critério Todos-Potenciais-Usos em relação ao critério Análise de Mutantes.

2.8 Considerações Finais

Diversos critérios de teste têm sido propostos e estudados com o objetivo de encontrar uma maneira sistemática para construir conjuntos de casos de teste que sejam eficazes em revelar defeitos no software a um baixo custo.

Estudos empíricos são conduzidos para comparar e avaliar os critérios de teste, definindo estratégias de teste para detectar o máximo de defeitos existentes no software e com o menor custo possível. Além disso, o desenvolvimento de ferramentas de teste que automatizam os critérios de teste contribuem para reduzir o esforço e aumentar a qualidade e produtividade da atividade de teste.

Nesse capítulo discutiram-se as principais características da atividade de teste, as fases necessárias para sua condução, e as técnicas e critérios de teste mais utilizados. Uma visão geral foi apresentada sobre técnicas de geração de dados de teste, com ênfase na geração de dados de testes sensíveis a erros.

O critério Análise de Mutantes e os Critérios Restritos foram descritos com maiores detalhes pois são os critérios mais utilizados nesse trabalho. Foram apresentadas algumas ferramentas que automatizam a atividade de teste de software. Estudos empíricos relacionados aos critérios Baseados em Fluxo de Dados e Análise de Mutantes foram descritos.

Devido à inexistência de estudos empíricos para comparar os Critérios Restritos e o critério Análise de Mutantes, e tendo em vista a eficácia desses critérios em revelar a presença de defeitos, um experimento foi conduzido para comparar esses critérios, o qual é descrito no próximo capítulo.

Capítulo 3

Critérios Restritos x Análise de Mutantes

Esse capítulo apresenta a comparação empírica realizada entre os Critérios Restritos e o critério Análise de Mutantes.

Para fazer essa comparação foi realizado um experimento. Os principais objetivos e passos seguidos durante esse experimento são detalhados. Uma análise dos resultados é apresentada considerando os fatores custo, eficácia e dificuldade de satisfação *(strength)*. Esse experimento também tornou possível uma comparação entre o critério baseado em fluxo de dados, Todos-Potenciais-Usos e o critério Análise de Mutantes.

Durante a aplicação do critério Análise de Mutantes foram observados alguns resultados com relação a mutantes equivalentes. Como mutantes equivalentes constituem uma grande limitação para a aplicação e automatização do critério Análise de Mutantes, esses resultados são apresentados no Capítulo 4. Os principais objetivos do experimento são apresentados a seguir:

- Comparar o critério Todos-Potenciais-Usos Restritos com o critério Análise de Mutantes considerando os fatores: custo, em termos do número de casos de teste necessários e eficácia, em termos de número de erros revelados;

- Avaliar o *strength* (dificuldade de satisfação) do critério Análise de Mutantes (AM) em relação aos critérios Todos-Potenciais-Usos (PU) e Todos-Potenciais-Usos Restritos (PU-R);
- Avaliar o *strength* (dificuldade de satisfação) do critério Análise de Mutantes, considerando apenas o operador de mutação ORRN (Relational Operator Replacement) em relação ao critério Todos-Potenciais-Usos Restritos utilizando as restrições BOR/BRO descritas no Capítulo 2.

3.1 Descrição do Experimento e Coleta dos Resultados

Para realizar o experimento, foi utilizada a ferramenta Proteum [Del93], que apóia a aplicação do critério Análise de Mutantes para o teste de programas na linguagem C. Para comparar os Critérios Restritos com o critério Análise de Mutantes foi escolhido o critério Todos-Potenciais-Usos Restritos, implementados pela ferramenta Poke-Tool [Ver97].

Esse experimento foi realizado utilizando um grupo formado por oito programas utilitários UNIX escritos na linguagem C, que são de domínio público: *cal, checkeq, col, comm, look, spline, tr, uniq*, descritos na Tabela 3.1. Tais programas fazem parte de um *benchmark* usado inicialmente por Wong [WMM94] e posteriormente por Vergilio [Ver97]. Os motivos para escolher esse conjunto de programas foram:

1. A existência de um conjunto de programas incorretos para os programas a serem testados, cujos erros encontram-se classificados em diferentes categorias, o que permite a análise da eficácia dos critérios;
2. A utilização dos mesmos conjuntos de restrições e testes gerados no experimento de Vergilio [Ver97], não sendo necessário repetir o experimento com o critério Todos-Potenciais-Usos Restritos.

Vergilio [Ver97] realizou uma comparação entre o critério Todos-Potenciais-Usos Restritos e seu correspondente critério estrutural Todos-Potenciais-Usos. Nesse experimento, restrições BOR/BRO foram associadas às associações

requeridas pelo critério Todos-Potenciais-Usos, e os resultados obtidos nesse trabalho são apresentados nesse capítulo e utilizados para comparação dos critérios.

Os programas testados são constituídos de um conjunto de unidades. Algumas características sobre cada programa, são apresentadas na Tabela 3.2: número de unidades, número de linhas de código, complexidade de McCabe, número de nós, número de predicados, número de predicados compostos e a porcentagem desses em relação ao número total de predicados.

Tabela 3.1. - Descrição dos Programas

Programas	**Objetivo**
Cal	Exibe um calendário para um determinado mês ou ano.
Checkeq	Informa os delimitadores ausentes ou desbalanceados e pares .EQ/.EN.
Comm	Compara dois arquivos ordenados linha a linha. Seleciona ou rejeita linhas comuns entre dois arquivos
Col	Filtra avanço de linhas inversas.
Look	Procura palavras em um dicionário ou linhas em uma lista ordenada
Tr	Copia a entrada padrão para a saída padrão e altera ou reduz caracteres repetidos no resultado
Spline	Toma pares de números como abcissas e ordenadas de uma função e produz um conjunto similar.
Uniq	Informa ou remove linhas adjacentes duplicadas

Tabela 3.2. - Principais Características dos Programas Utilizados

Programas	**N.° Unidades**	**N.° Linhas Código**	**Complexidade de McCabe**	**N.° Nós**	**N.° Pred.**	**N.° Pred. Compostos**	**(%)**
Cal	4	203	28	59	20	4	20
Checkeq	2	104	26	61	24	5	20,8
Comm	6	170	40	84	24	3	12,5
Col	7	301	65	118	37	5	13,5
Look	4	146	33	71	17	3	17,6
Tr	3	141	44	97	32	4	12,5
Spline	7	332	64	119	40	7	17,5
Uniq	6	140	34	67	21	3	14,3
Total	**37**	**1537**	**334**	**676**	**221**	**34**	**15,4**

Basicamente, o experimento conduzido é composto das seguintes etapas para cada um dos oito programas:

1. Geração dos Programas Mutantes;
2. Geração de conjuntos AM-adequados partindo dos conjuntos iniciais de casos de teste para cada programa;
3. Análise da Eficácia;
4. Análise dos Resultados.

3.1.1 Geração dos Programas Mutantes

Para geração dos mutantes foram utilizados todos os operadores disponíveis na ferramenta Proteum [Del93], os quais são descritos no Apêndice A.

3.1.2. Geração de conjuntos AM-adequados

A cobertura inicial do critério Análise de Mutantes é obtida utilizando-se os mesmos conjuntos de dados de teste utilizados por Vergilio [Ver97]. Os conjuntos utilizados são descritos a seguir:

- conjunto de dados de teste "ad-hoc": foi gerado manualmente por Vergilio [Ver97], fazendo-se uso da especificação do programa, do código fonte e do grafo de fluxo de controle, sem entretanto, aplicar nenhuma técnica de geração;
- conjunto de dados de teste aleatório: foi gerado por Wong [WMM94].

Os conjuntos PU-adequados e PU-R-adequados foram obtidos do experimento de Vergilio, que executou os seguintes passos utilizando a ferramenta Poke-Tool:

- geração dos elementos requeridos pelo critério PU;
- satisfação do critério PU e obtenção da cobertura: nesse passo foi estabelecido o conjunto PU-adequado, e as associações potenciais usos não executáveis foram determinadas;

- geração dos elementos requeridos pelo critério PU-R: foram gerados os elementos restritos utilizando-se restrições BOR/BRO;
- satisfação do critério PU-R: através dos dados utilizados no passo anterior e de dados adicionais gerados manualmente. Nesse passo foi estabelecido o conjunto PU-R-adequado e as associações potenciais uso restritas não executáveis foram determinadas.

Vergilio também utilizou os conjuntos de dados aleatórios gerados por Wong e verificou a cobertura para ambos os critérios PU e PU-R, obtendo também conjuntos PU-aleat e PU-R-aleat que contêm os dados de teste aleatórios que realmente contribuíram para aumentar a cobertura de ambos os critérios. No entanto, note que esses conjuntos não são adequados porque dados adicionais não foram gerados para a completa satisfação dos critérios.

Para gerar os conjuntos AM-adequados fez-se uso dos conjuntos "ad-hoc" e aleatórios, e submissão desses dados à ferramenta Proteum. Baseando-se nesses conjuntos e nas informações fornecidas pela ferramenta Proteum (mutantes vivos), foram adicionados novos casos de teste ao conjunto até obter-se um conjunto AM-adequado para cada programa.

A determinação de mutantes equivalentes é responsabilidade do testador, visto que a ferramenta Proteum utilizada não possui mecanismos que automatizem essa atividade. Os mutantes foram analisados para cada programa, e foram determinados os mutantes equivalentes; essa tarefa consumiu grande quantidade de esforço.

A Tabela 3.3 apresenta as informações referentes aos conjuntos adequados aos critérios Todos-Potenciais-Usos (PU), Todos-Potenciais-Usos Restritos (PU-R) e Análise de Mutantes (AM). São descritos os elementos requeridos e o número de casos de teste necessários para satisfazer os critérios comparados.

Essa Tabela também apresenta a cobertura obtida para os três critérios. Note que a cobertura apresentada não é 100% pois para todos os programas existiam elementos não executáveis (critérios PU e PU-R) e mutantes equivalentes (critério AM), que não foram descontados. Informações sobre

mutantes equivalentes e elementos não executáveis são apresentados na última coluna.

Tome-se como exemplo o programa *cal:* para o critério PU são requeridos 242 elementos, são necessários 12 casos de teste para satisfazê-lo, ou seja o conjunto PU-adequado possui 12 casos de teste, a cobertura obtida foi 71,49% e o número de elementos não executáveis foi 69, o que representa 28,51% do total dos 242 elementos requeridos; o critério PU-R requereu 488 elementos, 17 casos de teste para satisfazê-lo, a cobertura obtida é 63,11% e o número de elementos não executáveis é 180, ou seja, 36,89% dos elementos requeridos; e por último o critério AM, gerou 4324 mutantes, 32 casos de teste para satisfazê-lo, a cobertura obtida é 93% e o número de mutantes equivalentes é 309, 7% do total dos elementos requeridos.

Tabela 3.3. - Conjuntos adequados aos critérios PU, PU-R e AM

Programa	Critério	Nº Elem. Requeridos	Nº Casos de Teste	Cobertura (%)	Mutantes Equivalentes /Elementos não executáveis	
cal	PU	242	12	71,49	69	(28,51 %)
	PU-R	488	17	63,11	180	(36,89 %)
	AM	4324	32	93,00	309	(07,00 %)
checkeq	PU	582	76	80,41	114	(19,59 %)
	PU-R	1539	164	69,40	471	(30,60 %)
	AM	3075	74	93,00	214	(7,00 %)
col	PU	999	69	87,49	125	(12,51 %)
	PU-R	1807	75	77,75	402	(22,25 %)
	AM	6910	78	87,00	887	(13,00 %)
comm	PU	427	68	64,87	150	(35,13 %)
	PU-R	884	74	56,67	383	(43,33 %)
	AM	1938	47	85,00	281	(15,00 %)
look	PU	524	34	83,00	87	(17,00 %)
	PU-R	1045	40	76,36	247	(23,64 %)
	AM	2030	43	89,00	216	(11,00 %)
spline	PU	999	57	79,38	206	(20,62 %)
	PU-R	2352	62	59,35	956	(40,65 %)
	AM	12560	81	89,00	1426	(11,00 %)
tr	PU	1527	30	38,90	933	(61,10 %)
	PU-R	3040	35	33,75	2014	(66,25 %)
	AM	4422	72	82,13	790	(17,87 %)
uniq	PU	262	36	87,79	32	(12,21 %)
	PU-R	552	42	75,36	136	(24,64 %)
	AM	1623	36	90,00	160	(10,00 %)
Total PU		**5562**	**382**	**69,15**	**1716**	**(30,85%)**
Total PU-R		**11707**	**509**	**59,09**	**4789**	**(40,91%)**
Total AM		**36882**	**463**	**88,39**	**4283**	**(11,61%)**

3.1.3 Análise da Eficácia dos Critérios

Foram utilizados dois conjuntos de programas incorretos para verificar a eficácia dos critérios em teste. Os erros foram classificados em erros de domínio, erros de computação e erros de estrutura de dados.

As Tabelas 3.4 e 3.5 apresentam o número de versões incorretas para cada programa em cada categoria, para os dois conjuntos de programas gerados: Conjunto 1: o mesmo conjunto utilizado por Vergilio e Wong e Conjunto 2: gerado nesse experimento e cujos erros foram introduzidos aleatoriamente por alunos de um curso de graduação em computação.

Tabela 3.4 - Classificação dos erros introduzidos em cada programa - Conjunto 1

Programa	**Erro de Computação**	**Erro de Domínio**	**Erro de Estrutura de Dados**	**Total**
cal	12	7	1	20
Checkeq	13	9		22
Col	25	10	2	37
Comm.	8	11		19
look	10	9	1	20
spline	11	9		20
tr	13	6		19
uniq	9	9	1	19
Total	**101 (57,39%)**	**70 (39,77%)**	**5 (2,84%)**	**176**

Tabela 3.5 - Classificação dos erros introduzidos em cada programa – Conjunto 2

Programa	**Erro de Computação**	**Erro de Domínio**	**Erro de Estrutura de Dados**	**Total**
cal	5	4	1	10
checkeq	1	8	1	10
col	3	7		10
comm	6	4		10
look	3	7		10
spline	7	2	1	10
tr	4	5	1	10
uniq	2	7	1	10
Total	**31 (38,75%)**	**44 (55%)**	**5 (6,25%)**	**80**

As Tabelas 3.6 e 3.7 são tabelas de eficácia para os dois conjuntos de programas incorretos. Elas foram obtidas executando-se as versões incorretas com os conjuntos PU-adequados, PU-R-adequados e AM-adequados, e

comparando-se as saídas obtidas com as saídas dos programas originais. Assim, foram observados quais dados de teste revelaram os erros dos conjuntos de programas incorretos. Cada versão incorreta em ambos os conjuntos possui apenas um erro introduzido.

Na Tabela 3.6 são apresentadas informações referente ao Conjunto 1, que descreve os dados para os critérios PU, PU-R e AM, sendo que para os dois primeiros critérios os dados foram obtidos do experimento de Vergilio [Ver97]. Os erros revelados são apresentados para cada categoria. Como exemplo, para o programa *cal* e para o critério PU foram revelados: 12 erros de computação, 6 erros de domínio (dos 7 existentes) representados por 6/7, e 1 erro de estrutura de dados. As três últimas linhas apresentam os totais de números revelados pelos critérios PU, PU-R e AM, por exemplo para o critério PU foram revelados 100 (99%) dos 101 erros de computação existentes, 65 (92,86%) dos 70 erros de domínio e 4 (80%) dos 5 erros de estrutura de dados existentes; o total destes erros é apresentada na última coluna e significa que o critério PU revelou 169 (96,02%) dos 175 erros existentes.

A Tabela 3.7 apresenta a análise de eficácia para os programas do experimento utilizando o Conjunto 2.

Maiores detalhes sobre esse experimento podem ser encontrados em Soares [Soa00a]. No Apêndice B são apresentados os dados coletados em um dos programas do experimento.

Tabela 3.6 - Eficácia dos Critérios PU, PU-R e AM – Conjunto 1

Prog.	Critério	Erro de Comp.	Erro de Domínio	Erro de Estrut. de Dados	Total
cal	PU	12	6/7	1	19/20
	PU-R	12	7	1	20
	AM	12	7	1	20
checkeq	PU	12/13	8/9		20/22
	PU-R	12/13	8/9		20/22
	AM	13	9		22
col	PU	25	10	2	37
	PU-R	25	10	2	37
	AM	25	10	2	37
comm	PU	8	11		19
	PU-R	8	11		19
	AM	8	11		19
look	PU	10	9	1	20
	PU-R	10	9	1	20
	AM	10	9	1	20
spline	PU	11	6/9		17/20
	PU-R	11	6/9		17/20
	AM	11	9		20
tr	PU	13	6		19
	PU-R	13	6		19
	AM	13	6		19
Uniq	PU	9	9	0/1	18/19
	PU-R	9	9	0/1	18/19
	AM	9	9	0/1	18/19
N.º Rev. PU		100 /101	65/70	4/5	169/176
Rev. PU %		99	92,86	80	96,02
N.º Rev. PU-R		100 /101	66/70	4/5	170/176
Rev. PU-R %		99	94,29	80	96,59
N.º Rev AM		101	70	4/5	175/176
Rev. AM %		100	100	80	99,43

Tabela 3.7 - Eficácia dos Critérios PU, PU-R e AM – Conjunto 2

Prog.	Critério	Erro de Comp.	Erro de Domínio	Erro de Estrut. de Dados	Total
cal	PU	5	3/4	1	9/10
	PU-R	5	4	1	10
	AM	5	4	1	10
checkeq	PU	1	5/8	0/1	6/10
	PU-R	1	5/8	0/1	6/10
	AM	1	6/8	1	8/10
col	PU	3	7		10
	PU-R	3	7		10
	AM	3	7		10
comm	PU	4/6	4		8/10
	PU-R	4/6	4		8/10
	AM	5/6	4		9/10
look	PU	3	7		10
	PU-R	3	7		10
	AM	3	7		10
spline	PU	7	2	1	10
	PU-R	7	2	1	10
	AM	7	2	1	10
tr	PU	3/4	5/6		8/10
	PU-R	3/4	5/6		8/10
	AM	4	6		10
uniq	PU	2	5/7	0/1	7/10
	PU-R	2	5/7	0/1	7/10
	AM	2	6/7	1	9/10
N.º Rev. PU		28/30	38/46	2/4	68/80
Rev. PU %		93,33	82,60	50	85
N.º Rev. PU-R		28/30	39/46	2/4	69/80
Rev. PU-R %		93,33	84,78	50	86,25
N.º Rev AM		30	42/46	4	76/80
Rev. AM %		100	91,30	100	95

3.2 Análise dos Resultados

Nesta seção é realizada uma análise dos resultados obtidos do experimento de acordo com os objetivos estabelecidos no início deste capítulo.

3.2.1 Análise do *Strength* dos Critérios

Para analisar a dificuldade de satisfação (*strength)* do critério Análise de Mutantes (AM) em relação aos critérios Todos-Potenciais-Usos (PU) e Todos-

Potenciais-Usos Restritos (PU-R), os conjuntos de casos de teste PU-adequados e PU-R adequados foram aplicados para o critério AM. A Tabela 3.8 apresenta a cobertura (escore de mutação) obtida para o critério AM através da utilização dos conjuntos PU-adequados (T_{PU}) e PU-R-adequados (T_{PU-R}), a última coluna dessa tabela apresenta a cobertura (escore de mutação) obtida para o conjunto AM-adequado. O escore de mutação apresentado nessa tabela para os conjuntos AM-adequados não são 100% pois os mutantes equivalentes não foram descontados, desse modo, para o programa *cal* faltou cobrir 3% dos mutantes usando o conjunto PU-adequado e 1% usando o conjunto PU-R-adequado.

A Tabela 3.9 apresenta a cobertura obtida para o critério AM usando os conjunto PU-aleat e PU-R-aleat obtidos por Vergilio.

Tabela 3.8 - Escore de mutação usando os conjuntos PU-adequados e PU-R adequados - Conjunto de dados de teste "ad-hoc"

Programas	**Escore de mutação para os conjuntos de casos de teste**		
	T_{PU}	T_{PU-R}	T_{AM}
Cal	90.0	92.0	93.0
Checkeq	88.0	90.0	93.0
Col	79.0	81.0	85.0
Comm	82.0	82.0	87.0
Look	83.0	83.0	89.0
Spline	85.0	85.0	89.0
Tr	64.0	71.0	82.0
Uniq	88.0	89.0	90.0
Média	**82.3**	**84.1**	**88.5**

Tabela 3.9 - Escore de mutação usando os conjuntos PU-aleat e PU-R-aleat - Conjunto de dados de teste aleatório

Programas	**Escore de mutação para os conjuntos de casos de teste**		
	T_{PU}	T_{PU-R}	T_{AM}
Cal	83.0	91.0	93.0
Checkeq	79.0	79.0	93.0
Col	81.0	81.0	85.0
Comm	81.0	81.0	87.0
Look	80.0	80.0	89.0
Spline	79.0	79.0	89.0
Tr	56.0	56.0	82.0
Uniq	81.0	81.0	90.0
Média	**77.5**	**78.5**	**88.5**

Através dos resultados obtidos é possível fazer as seguintes observações:

- Conjuntos de dados de teste "ad-hoc" – (conjuntos adequados)
  - A média do escore de mutação obtido para o critério AM através da utilização do conjunto PU-R-adequado foi 1,8% maior que a obtida através da utilização do conjunto PU-adequado;
  - A média do escore de mutação obtido para o conjunto AM-adequado foi 6,2% maior que a obtida através da utilização do conjunto PU-adequado e 4,4% maior que a obtida através da utilização do conjunto PU-R-adequado;
  - Entretanto, o escore de mutação obtido para o critério AM a partir dos conjuntos PU-adequados (82,3%) e PU-R-adequados (84,1%) foi bastante alto, considerando que o escore de mutação do conjunto AM-adequado foi 88,5%, apenas 6,2% dos mutantes não fora cobertos pelo critério PU e 4,4% não foram cobertos pelo critério PU-R;
  - O menor escore de mutação obtido para o critério AM a partir dos conjuntos PU-adequados (64%) e PU-R-adequados (71%) foi para o programa *tr*, uma explicação para isso é o grande número de variáveis e constantes que esse programa possui. Para esse programa o escore

de mutação obtido utilizando o conjunto PU-R-adequado também foi o maior com relação ao escore do critério PU, isso se deve ao fato que nenhuma restrição referente às variáveis foram utilizadas pelo critério PU-R;

- O maior escore de mutação obtido para o critério AM a partir dos conjuntos PU-adequado (90%) e PU-R-adequado (92%) foi para o programa *cal*, o motivo é o número reduzido de variáveis e constantes existentes nesse programa.

- Conjuntos de dados de teste aleatórios
  - A média do escore de mutação obtido para o conjunto AM-adequado foi 11% maior que a obtida através da utilização do conjunto PU-aleat e 10% maior que a obtida através da utilização do conjunto PU-R-aleat;
  - O menor escore de mutação obtido para o critério AM a partir dos conjuntos PU-adequados (56%) e PU-R-adequados (56%) foi novamente para o programa *tr*;
  - O maior escore de mutação obtido para o critério AM a partir dos conjuntos PU-adequado (83%) e PU-R-adequado (91%) também foi para o programa *cal*.

A média do escore de mutação obtida através do conjunto "ad-hoc" foi maior que a obtida através do conjunto aleatório: 5,6% maior para o critério PU-R e 4,8% maior para o critério PU.

A cobertura para o critério AM, utilizando-se os conjuntos PU-aleat e PU-R-aleat foi menor do que utilizando-se os conjuntos PU-adequados e PU-R-adequados. Pode-se concluir que satisfazer os critérios PU e PU-R implica em conseguir uma melhor cobertura para o critério AM, e que é mais fácil satisfazer o critério AM dado que o critério PU-R foi satisfeito do que dado que o critério PU foi satisfeito. Note que isso ocorreu para todos os programas e para o programa *tr* foi maior essa diferença.

3.2.2 Análise do Custo dos Critérios

A Tabela 3.10 apresenta um resumo de informações sobre o custo para os critérios PU, PU-R e AM, esses dados foram obtidos da Tabela 3.3 descrita anteriormente. A Tabela 3.10 é descrita da seguinte forma: a segunda coluna apresenta a *média de número de casos de teste* para todos os programas do experimento, na terceira coluna o program *checkeq* é excluído do conjunto, e na quarta são excluídos os programas *checkeq e comm*.

Tabela 3.10 - *Custo dos critérios* PU, PU-R e AM

Critério	Conjunto de programas completo	Conjunto de programas sem o prog. *checkeq*	Conjunto de programas sem os prog. *checkeq e comm*
	Nº Casos de Teste	Nº Casos de Teste	Nº Casos de Teste
PU	382	306	238
PU-R	509	345	271
AM	463	389	342

A média do custo obtido para todos os programas apresentado na segunda coluna, mostra que o custo do critério AM foi 21,20% superior ao critério PU e 9,94% inferior ao critério PU-R. O programa *checkeq* é o maior responsável pela diferença de custo entre esses critérios e a explicação para isso é a estrutura desse programa, que possui uma grande quantidade de *variáveis* e *comandos if's*, que faz com que um grande número de elementos sejam requeridos pelo critério PU; Outra explicação, é o tipo de restrição (BOR) usada no critério PU-R que se aplica a predicados compostos. A maioria dos comandos *if's* do programa *checkeq* possui predicados compostos. Esses fatos ocasionam o aumento do número de casos de testes necessários. O programa *comm* apresentou um baixo número de casos de teste para o critério AM porque a maioria dos casos de testes inseridos mataram vários mutantes, o que não ocorreu com os outros programas que necessitaram de vários casos de testes para matar apenas um mutante.

A terceira coluna mostra que excluindo somente o programa *checkeq*, o custo obtido para satisfazer o critério AM é 27,12% superior ao critério PU e 12,75% superior ao critério PU-R.

Observe na quarta coluna da Tabela 3.10, que ao excluir os programas *checkeq* e *comm*, o custo obtido para satisfazer o critério AM é 43,70% superior ao critério PU e 26,20% superior ao critério PU-R.

Pode-se concluir através dos dados analisados e desconsiderando o programa *checkeq*, que de modo geral o critério AM é mais caro.

3.2.3 Eficácia dos Critérios PU-R e AM

A análise apresentada a seguir refere-se à eficácia obtida através da utilização do Conjunto 1, os dados dos conjuntos PU-adequados, PU-R-adequados e AM-adequados foram utilizados como descritos anteriormente e apresentados na Tabela 3.6:

- o critério AM foi o mais eficaz e revelou 3,41% e 2,84% mais erros que os critérios PU e PU-R respectivamente;
- o critério AM revelou 1% mais erros de computação que os critérios PU e PU-R;
- o critério AM revelou 7,14 % e 5,71% mais erros de domínio que PU e PU-R respectivamente;
- o critério PU-R revelou 1,43% mais erros de domínio que o critério PU;
- Não houve diferença entre os critérios quando considerados apenas os erros em estrutura de dados.

A análise apresentada a seguir refere-se ao Conjunto 2, os dados são descritos na Tabela 3.7 apresentada anteriormente:

- o critério AM foi o mais eficaz e revelou 10% e 8,75% mais erros que os critérios PU e PU-R respectivamente;
- o critério AM revelou 6,67% mais erros de computação que os critérios PU e PU-R;
- o critério AM revelou 8,70 % e 6,52% mais erros de domínio que PU e PU-R respectivamente;

- o critério AM revelou 100% mais erros de estrutura de dados que PU e PU-R;
- o critério PU-R revelou 2,18% mais erros de domínio que o critério PU.

Através desses dados pode-se concluir que entre os três critérios, o critério AM mostrou-se o mais eficaz independentemente do tipo de erro considerado, em seguida está o critério PU-R e depois o critério PU.

3.2.4 Análise do Operador ORRN

O operador ORRN (Relacional Operador Replacement) faz parte do grupo de operadores de mutação da ferramenta Proteum e visa a substituir todas as ocorrências de operadores relacionais por outros operadores relacionais. No exemplo mostrado a seguir o operador relacional != é substituído pelo operador relacional ==.

Comando original -> *if (x != y)*
Comando alterado -> *if (x == y)*

As restrições associadas aos elementos requeridos pelo critério Todos-Potenciais-Usos (PU) para o estabelecimento dos elementos restritos requeridos pelo critério Todos-Potenciais-Usos Restritos (PU-R) foram restrições BOR/BRO. Essas restrições foram descritas no Capítulo 2 e têm como objetivo revelar a presença de defeitos em predicados se eles estiverem presentes.

Como o operador relacional ORRN está associado a esse tipo de defeito foi realizada uma análise especial do strength (dificuldade de satisfação) do critério Análise de Mutantes considerando apenas o operador ORRN, utilizando-se o conjunto de dados PU-R-adequado.

A Tabela 3.11 apresenta os resultados: para cada programa é descrito o número de mutantes gerados, o número de mutantes equivalentes determinados, o número de dados de teste do conjunto PU-R-adequado representado por $T_{PU\text{-}R}$, a cobertura (escore de mutação) obtida para o critério Análise de Mutantes através da utilização do conjunto $T_{PU\text{-}R}$, e nas duas últimas colunas é apresentado o número de dados de teste do conjunto AM-adequado representado por T_{AM} e a

cobertura obtida para o critério AM. As coberturas apresentadas foram obtidas incluindo-se os mutantes equivalentes, por isso não são 100%.

Tabela 3.11 - Escore de mutação obtida para o operador ORRN usando o conjunto PU-R adequado

Programa	Mutantes		T_{PU-R}	Cobertura (%)	T_{AM}	Cobertura (%)
	Gerados	Equivalentes				
Cal	110	13 (12%)	11	85	14	88
Checkeq	135	13 (10%)	28	82	38	90
Col	200	55 (27%)	22	73	22	73
Comm.	115	18 (16%)	15	81	18	84
Look	60	10 (17%)	9	80	11	83
Spline	210	40 (19%)	16	78	21	81
Tr	140	27 (29%)	11	65	21	81
Uniq	95	13 (14%)	13	85	13	85

Na maioria dos casos, exceto para dois programas *col* e *uniq*, foram necessários casos de teste adicionais para satisfazer o critério Análise de Mutantes. Esses resultados mostram que para esses programas restrições BOR/BRO não incluem o operador ORRN. Esse fato pode ser explicado devido a presença de elementos (associações) não executáveis requerido pelo critério Todos-Potenciais-Usos Restritos.

3.3 Considerações Finais

Esse capítulo apresentou um experimento empírico realizado com a finalidade de comparar o critério Todos-Potenciais-Usos Restritos e o critério Análise de Mutantes, considerando os fatores eficácia, custo e *strength*.

Os resultados demonstraram que o critério Análise de Mutantes possui um maior custo de aplicação no que se refere ao número de casos de teste, exceto quando o programa em teste possui estruturas que podem aumentar o número de elementos requeridos para um critério estrutural: o programa *checkeq* exemplifica isso. Com relação à eficácia, observou-se que os Critérios Restritos se mostraram

mais eficazes que o correspondente critério estrutural porém menos eficazes que o critério Análise de Mutantes.

Através da avaliação do *strength* (dificuldade de satisfação) do critério Análise de Mutantes em relação ao critério Todos-Potenciais-Usos Restritos, pode-se observar que é mais fácil satisfazer o critério Análise de Mutantes tendo sido satisfeito o critério Todos-Potenciais-Usos Restritos do que tendo sido somente satisfeito o critério Todos-Potenciais-Usos.

Comparando-se os resultados obtidos, utilizando-se os conjuntos "ad-hoc" e aleatórios, verifica-se que é importante satisfazer os critérios, isso implica na obtenção de uma maior cobertura e pode implicar num maior número de erros revelados. Portanto, em casos onde a confiabilidade requerida é alta, a geração aleatória pode não ser suficiente, e satisfazer o critério é necessário.

Foi realizada uma análise do *strength* (dificuldade de satisfação) do critério Análise de Mutantes considerando apenas o operador ORRN, com relação ao critério Todos-Potenciais-Usos Restritos, dado o fato de que o operador ORRN assim como as restrições BOR/BRO utilizadas pelo critério Todos-Potenciais-Usos Restritos (PU-R) têm o objetivo de encontrar erros em predicados. Nota-se que as restrições BOR/BRO não incluem a mutação do operador relacional na presença de caminhos não executáveis.

Considerando os resultados aqui obtidos é apresentada a seguir uma possível estratégia de aplicação para os critérios:

1. **Geração inicial de um conjunto de dados de teste**: esses dados podem ser obtidos através de uma estratégia aleatória ou aplicando-se um critério de teste funcional. A segunda abordagem pode ser mais interessante pois casos de teste poderiam ser gerados considerando também a saída esperada, o que facilitaria a avaliação dos testes realizados.

1. **Aplicação de um critério de teste**, isso envolve:

    2.1 geração dos elementos requeridos;

2.2 obtenção da cobertura inicial utilizando-se o conjunto derivado no passo 1;

2.3 geração de dados adicionais, nesse passo pode-se utilizar qualquer princípio, mas é necessário a determinação dos elementos requeridos não executáveis no caso do critério ser estrutural e dos mutantes equivalentes para o critério Análise de Mutantes, baseado em erros.

Sugere-se que no passo 2 seja utilizada uma ferramenta de apoio à aplicação do critério, e que os critérios sejam utilizados seguindo a relação empírica apontada pelos resultados apresentados nesse capítulo e seguindo a confiabilidade requerida para o programa. A ordem seria a seguinte:

- 1º Critérios Baseados em Fluxo de Controle - Todos-Nós, Todos-Ramos, etc;
- 2º Critérios Baseados em Fluxo de Dados - Critério Todos-Potenciais-Usos, Todos-Potenciais-Du-Caminhos, etc;
- 3º Critérios Restritos - Todos-Potenciais-Usos Restritos, Todos-Potenciais-Du-Caminhos Restritos, etc;
- 4º Critério Análise de Mutantes.

Capítulo 4

Mutantes Equivalentes

No teste de mutação um conjunto de casos de teste é avaliado para o teste de um programa em função da sua capacidade de diferenciar esse programa de outros programas, semelhantes a ele, chamados de mutantes. Os mutantes gerados e executados com o conjunto de casos de teste podem ter um dos dois comportamentos: se um mutante apresenta um resultado diferente do programa original ele é considerado morto, por outro lado se o mutante apresenta comportamento igual ao programa original para todas as entradas ele é um mutante equivalente.

A análise dos mutantes equivalentes é o passo que requer mais intervenção humana, e um dos maiores obstáculos para a aplicação do teste de mutação. Sem determinar todos os mutantes equivalentes o escore de mutação nunca será 100%. Assim o testador não terá completa confiança no programa e nos dados de teste. Pior, o testador será incapaz de saber se os mutantes restantes são equivalentes ou se o conjunto de teste é insuficiente. Determinar mutantes equivalentes manualmente consome muito tempo, o que contribui para elevar o custo do teste de mutação.

Em geral, o problema de resolver se dois programas são equivalentes é indecidível; essa limitação teórica não significa que o problema deva ser abandonado por não ter solução. Na verdade alguns métodos e heurísticas têm

sido propostos para determinar a equivalência de programas em uma grande porcentagem dos casos de interesse [BS79, Bud81].

Esse capítulo é dedicado à questão de mutantes equivalentes. Inicialmente são apresentados os principais resultados relacionados ao experimento de aplicação do critério Análise de Mutantes descrito no capítulo anterior, referentes aos mutantes equivalentes determinados.

Um estudo sobre os principais operadores que geraram mutantes equivalentes é apresentado; esse estudo pode ajudar a identificar os mutantes equivalentes.

Após isso, um resumo de técnicas que podem ser utilizadas para automatizar a determinação de mutantes equivalentes é apresentado.

4.1 Aplicação do Critério Análise de Mutantes

Na Tabela 4.1 são apresentadas informações referentes à aplicação do critério Análise de Mutantes aos programas: número de mutantes gerados, mortos e equivalentes; e na última coluna é apresentada a cobertura, sem considerar os mutantes equivalentes. Esses dados demonstram o grau de complexidade dos programas considerados no experimento, onde é possível observar que:

- O programa *spline* teve o maior número de mutantes gerados 12560;
- O programa *uniq* teve o menor número de mutantes gerados 1623;
- Os programas que tiveram o maior número de mutantes mortos e consequentemente o menor número de mutantes equivalentes (em porcentagem) foram os programas *cal e checkeq;*
- O programa que teve o menor número de mutantes mortos e consequentemente o maior número de mutantes equivalentes foi o programa *tr,* devido ao grande número de variáveis e constantes existentes nesse programa;
- A média de mutantes equivalentes determinados no experimento representa 11,65% dos mutantes gerados e não deve ser desprezada

pois grande quantidade de esforço foi gasto na atividade de determinação.

Em média foram gastas 240 horas para essa tarefa, foram analisados pelo menos 4.298 mutantes. Pode-se concluir que é muito importante oferecer mecanismos que auxiliem essa tarefa. Por isso, as seções seguintes tratam do problema de identificar e determinar automaticamente mutantes equivalentes.

Tabela 4.1 - Aplicação do critério AM

Programa	**Número de Mutantes**			**Cobertura**
	Gerados	**Mortos**	**Equivalentes**	
Cal	4334	4015	309 (7%)	93.00
Checkeq	3075	2861	214 (7%)	93.00
Col	6910	6023	887 (13%)	87.00
Comm	1938	1652	281 (15%)	85.00
Look	2030	1814	216 (11%)	89.00
Spline	12560	11134	1426 (11%)	89.00
Tr	4422	3632	790 (18%)	82.00
Uniq	1623	1463	160 (10%)	90.00
Total	**36892**	**32594**	**4298 (11,65%)**	**88.35**

4.2 Identificação x Operadores da Proteum

A Tabela 4.2 apresenta os totais de mutantes gerados e equivalentes determinados para todos os operadores da ferramenta Proteum, para os programas em teste: *cal, checkeq, col, comm, look, spline, tr e uniq*. Através desses dados podemos observar que:

- Os quatro operadores que geraram o maior número de mutantes equivalentes em relação ao número total de mutantes gerados em porcentagem, em ordem (do maior para o menor) foram: OCOR, VDTR, OLBN, e OEBA.
- Os quatro operadores que geraram o menor número de mutantes equivalentes em relação ao número total de mutantes gerados em

porcentagem, em ordem (do menor para o maior) foram: STRP, STRI, SSWM e OABA.

Tabela 4.2 - Informações sobre cada operador da Ferramenta Proteum para os programas do experimento

Operador	Mutantes			Operador	Mutantes		
	gerados	equivalentes	%		gerados	equivalentes	%
STRP	941	12	1.28	OLAN	241	63	26.14
STRI	312	5	1.59	OLBN	144	54	37.5
SSDL	930	71	7.63	OLLN			
SRSR	1116	49	4.39	OLRN	294	19	6.46
SGLR				OLSN	96	2	2.08
SCRB	25	8	32	ORAN	1034	171	16.54
SBRC				ORBN	600	135	22.5
SBRn				ORLN	426	59	13.85
SCRn				ORRN	1065	189	17.75
SWDD	33	9	27.27	ORSN	400	90	22.5
SDWD				OSAA			
SMTT				OSAN			
SMTC	65	5	7.69	OSBA			
SMVB	66	7	10.61	OSBN			
SSWM	62	1	1.61	OSEA			
OAAA	123	6	4.88	OSLN			
OAAN	591	28	4.74	OSRN			
OABA	51	1	1.96	OSSA			
OABN	249	17	6.83	OSSN			
OAEA	21	2	9.52	Ouor			
OALN	334	13	3.89	OLNG			
OARN	1002	43	4.29	OCNG	242	10	4.13
OASA				OBNG	37	1	2.70
OASN	166	6	3.61	OIPM	34	11	32.35
OBAA	15	3	20	OCOR	312	300	96.15
OBAN	20	6	30	Vsrr	10682	760	7.11
OBBA				Varr	153	18	11.76
OBBN	44	6	13.64	Vtrr	61	15	24.59
OBEA				Vprr	324	20	6.17
OBLN	44	4	9.09	VSCR	271	56	20.66
OBRN	132	10	7.58	VDTR	1818	692	38.06
OBSA				VTWD	1212	107	8.83
OBSN	44	2	4.55	CRCR	2273	106	4.66
OEAA	1224	220	17.97	Cccr	3139	473	15.07
OEBA	624	211	33.81	Ccsr	2715	104	3.83
OESA	416	49	11.78	Oido	241	17	7.05

A Tabela 4.3 apresenta os quatro operadores que mais geraram mutantes equivalentes por programa do experimento.

Tabela 4.3 - Operadores que geraram o maior número de mutantes equivalentes por programa

Programa	1º	2º	3º	4º
Cal	VDTR	Cccr	OEBA	OEAA
Checkeq	VDTR	OEBA/ ORAN/ ORRN	OEAA	OLAN/ ORBN
Col	OCOR	Cccr	VDTR	VSRR
Comm	Cccr	OEBA/ VDTR	OEAA	ORAN/ORBN
Look	VDTR	Cccr	OCOR	OEBA
Spline	Vsrr	VDTR	CRCR	OEAA
Tr	Vsrr	Cccr	VDTR	OEBA
Uniq	OCOR	VDTR	Cccr	OEBA/ORRN

A seguir são apresentados, com uma breve explicação, os operadores da ferramenta Proteum que geraram o maior e menor número de mutantes equivalentes em relação ao total de mutantes gerados para os programas em teste. Serão apresentados exemplos das mutações equivalentes ocorridas nos programas do experimento para os operadores que geraram o maior número de mutantes equivalentes.

4.2.1 – Operadores que geraram o maior número de mutantes equivalentes

OCOR - Cast Operator by Cast Operator

Este operador pertence ao grupo das mutações de operadores. Os operadores de *"cast"* que envolvem os tipos primitivos da linguagem (int, char, long, float, etc) são trocados por operadores de cast com todos os outros tipos primitivos da linguagem. A Figura 4.1 exemplifica esse operador, onde é utilizada a rotina *canon* do programa *look*. O objetivo da função *tolower()* é devolver o equivalente minúsculo da variável *c* se *c* for uma letra, caso contrário *c* será devolvida sem alteração. A mutação do valor de retorno da função tolower() que é um tipo *int* por um tipo *float* nesse caso produzirá uma saída do programa igual ao programa original pois um tipo *float* pode armazenar sem problemas um tipo *int;* portanto, esse mutante é equivalente ao original.

```
canon(char * src, char * copia)
{ int cnt;
  char c;
  for (cnt = len + 1; (c = *src++) && cnt; --cnt)
     if (!dict || isalnum(c))
          *copia++ = fold && isupper(c) ? (int)tolower(c) : c;
■         *copia++ = fold && isupper(c) ? (float)tolower(c) : c;
     *copia = EOS;
}
```

Figura 4.1 - Exemplo da equivalência do operador OCOR ao programa original.

VDTR - Domain Traps

Este operador pertence ao grupo de mutações de variáveis, onde cada referência escalar é substituída pelas chamadas às funções *trap_on_zero (e)*, *trap_on_positive (e)* e *trap_on_negative(e)*. Estas funções abortam a execução dos mutantes caso a expressão tenha valor zero, positivo ou negativo, respectivamente. Caso contrário retornam o valor da expressão.

A Figura 4.2 exemplifica esse operador, onde é utilizada a rotina *main* do programa *cal*. Nesse exemplo a variável *argc* é um parâmetro que contém o número de argumentos da linha de comando e é um inteiro, *argc* é sempre pelo menos 1 porque o nome do programa é qualificado como primeiro argumento, e não é possível tornar a expressão argc < 0 verdadeira, e fazer o mutante se comportar diferentemente do programa original.

```
main(int argc, char *argv[ ])
{
        register y, i, j;
        int m;
        if (argc == 2)
■       if ((TRAP_ON_NEGATIVE(argc) == 2))
          goto xlong;
        if(argc < 2) {
                time_t t;
                struct tm *tm;
                t = time(0);
                tm = localtime(&t);
                m = tm->tm_mon + 1;
                y = tm->tm_year + 1900;
        } else {
                m = atoi(argv[1]);
                if(m<1 || m>12) {
                        fprintf(stderr, "cal: %s: Bad month.\n", argv[1]);
                        exit(1);
                }
                y = atoi(argv[2]);
                if(y<1 || y>9999) {
                        fprintf(stderr, "cal: %s: Bad year.\n", argv[2]);
                        exit(2);
                }
        }
        exit(0);
}
```

Figura 4.2 - Exemplo da equivalência do operador VDTR ao programa original.

OLBN - Logical Operator by Bitwise Operator

Este operador pertence ao grupo de mutações de operadores, onde cada operador lógico é substituído por um operador bitwise.

A Figura 4.3 exemplifica esse operador, onde é utilizada a rotina *main* do programa *cal*. O operador lógico || no comando if(m<1 || m>12) é trocado pelo operador bitwise $\wedge$, *independentemente* de quais valores a variável *m* possuir o resultado será equivalente ao resultado do programa original; como exemplo, se a variável *m* possuir o valor -1 o comando será executado e a mensagem indicando

" Bad month, -1" será impressa porque para os dois programas 0 || 1 e 0 ∧ 1 será 1.

```
main(int argc, char *argv[ ])
{       register y, i, j;
        int m;
        if (argc == 2)
                goto xlong;
        if(argc < 2) {
                time_t t;
                struct tm *tm;
                t = time(0);
                tm = localtime(&t);
                m = tm->tm_mon + 1;
                y = tm->tm_year + 1900;
        } else {
                m = atoi(argv[1]);
                if(m<1 || m>12) {
■               if(m<1 ∧ m>12) {
                        fprintf(stderr, "cal: %s: Bad month.\n", argv[1]);
                        exit(1);
                }
                y = atoi(argv[2]);
                if(y<1 || y>9999) {
                        fprintf(stderr, "cal: %s: Bad year.\n", argv[2]);
                        exit(2);
        }}
        printf("   %s %u\n", smon[m-1], y);
        printf("%s\n", dayw);
        cal(m, y, string, 24);
        for(i=0; i<6*24; i+=24)
                pstr(string+i, 24);
        exit(0);}
```

Figura 4.3 - Exemplo da equivalência do operador OLBN ao programa original.

OEBA - Plain Assignment by Bitwise assignment

Este operador pertence ao grupo das mutações de operadores e troca atribuição plana por atribuição bitwise.

A Figura 4.4 exemplifica esse operador, onde é utilizada a rotina *main* do programa cal. A atribuição plana = do comando for (i=0; i < 6*24; i+=24) é trocada

pela atribuição bitwise &=, e o resultado dessa atribuição para a variável *i* continua sendo 0, portanto existe equivalência entre esses programas.

```
main(int argc, char *argv[ ])
{
        register y, i, j;
        int m;
        if (argc == 2)
                goto xlong;
        if(argc < 2) {
                time_t t;
                struct tm *tm;
                t = time(0);
                tm = localtime(&t);
                m = tm->tm_mon + 1;
                y = tm->tm_year + 1900;
        } else {
                m = atoi(argv[1]);
                if(m<1 || m>12) {
                        fprintf(stderr, "cal: %s: Bad month.\n", argv[1]);
                        exit(1);
                }
                y = atoi(argv[2]);
                if(y<1 || y>9999) {
                        fprintf(stderr, "cal: %s: Bad year.\n", argv[2]);
                        exit(2);
                }
        }
        printf("   %s %u\n", smon[m-1], y);
        printf("%s\n", dayw);
        cal(m, y, string, 24);
        for(i=0; i<6*24; i+=24)
■       for(i&=0; i<6*24; i+=24)
                pstr(string+i, 24);
        exit(0);}
```

Figura 4.4 - Exemplo da equivalência do operador OEBA ao programa original.

4.2.2 – Operadores que geraram o menor número de mutantes equivalentes

STRP - Trap on Statement Execution

Este operador pertence ao grupo das mutações de comando, o objetivo deste comando é revelar erros de código que possuem caminhos não executáveis (código não alcançável). Todos os comandos são substituídos por uma função *trap_on_stat()* que quando executada termina a execução do programa mutante, mostrando que ele é alcançável.

Exemplo:

```
if(m<1 || m>12) {
■ TRAP_ON_STAT( );
      fprintf(stderr, "cal: %s: Bad month.\n", argv[1]);
      exit(1);
  }
```

STRI - Trap on if Condition

Este operador pertence ao grupo das mutações de comando, e visa a garantir que a condição de cada comando *if* seja avaliada, pelo menos uma vez pelo ramo verdadeiro e pelo ramo falso. Cada comando do tipo *if (e) S* gera dois mutantes: *if (trap_on_true(e))* e *if (trap_on_false(e))*. Ao ser executada *trap_on_true (trap_on_false)*, o programa mutante é interrompido se a expressão avaliada é verdadeira (falsa). De modo contrário, a função retorna o valor da expressão.

Exemplo:

```
  if (m<1 || m>12) {
■ if (TRAP_ON_TRUE(m<1 || m>12)) {
      fprintf(stderr, "cal: %s: Bad month.\n", argv[1]);
      exit(1);
  }
```

SSWM - Switch Statement Mutation

Este operador pertence ao grupo das mutações de comando, e substitui o comando *switch (e)* pelo comando *switch (trap_on_case(e,a))*. Esta função aborta a execução do programa mutante quando *a* = = *e*, sendo que *a* é cada um dos rótulos utilizados nos *case's* do comando. Por isso, para matar os programas mutantes desse operador a expressão *e* deve ser avaliada pelo menos uma vez para cada valor utilizado nos *case's*. o valor da expressão.

Exemplo:

```
                case '?':;
■               case '?': TRAP_ON_CASE( );
```

OABA - Arithmetic Assignment by Bitwise Assignment

Este operador pertence ao grupo das mutações de operador, e visa substituir todos os operadores aritméticos de atribuição pelos operadores bitwise de atribuição.

Exemplo:

```
  total += soma;
■ total ^= soma;
```

Os operadores que geraram o menor número de mutantes equivalentes foram incluídos na ferramenta Proteum para se garantir a inclusão do critério Todos-Ramos. A maioria dos mutantes gerados por esses operadores são facilmente cobertos. Se uma estratégia incremental de aplicação dos critérios tal como a descrita no capítulo anterior for utilizada, esses operadores poderão ser desconsiderados, reduzindo o número de execuções do programa e consequentemente o custo do critério AM.

4.3 Determinação

Provar que um mutante é equivalente é uma tarefa complicada e árdua que exige uma análise e entendimento completo do programa. Budd e Angluin [BA82] estudaram o relacionamento entre equivalência e geração de dados de teste. Eles

provam que se há um procedimento computacional para gerar dados de teste adequado para o teste de mutação para um programa, existe também um procedimento computacional para checar se um programa é equivalente ao mutante e vice-versa. Eles também mostram que, como regra geral, não existe nenhum desses procedimentos computacionais. Desta forma não existe uma solução algorítmica completa para o problema de equivalência de mutação. Isto é, determinar a equivalência entre dois programas ou dois mutantes é um problema indecidível.

Felizmente os programas equivalentes tem duas vantagens sobre o problema geral de equivalência. Primeiro programas mutantes são muito semelhantes aos programas originais, existe apenas uma pequena mudança sintática realizada através de um operador de mutação. Budd e Angluin descrevem mutantes como vizinhos dos programas originais. Pesquisadores utilizam esse fato para desenvolver técnicas e heurísticas para determinar mutantes equivalentes [OC94, OJ97]. A segunda vantagem é que o teste de software é inerentemente uma ciência imperfeita; assim, os resultados parciais são de grande importância.

Acree [Acr80] observou que é possível prevenir as pessoas de cometerem erros ao marcarem mutantes equivalentes. Através de um estudo onde existiam 50 mutantes, metade desses sendo equivalentes, Acree verificou que as pessoas julgam corretamente os mutantes equivalentes em 80% do tempo. Em 12% do tempo as pessoas marcam mutantes equivalentes como não equivalentes (erro do tipo 2) e marcam mutantes não equivalentes como mutantes equivalentes (erro do tipo 1) em 8% do tempo. Como o erro do tipo 2 ocorre durante uma fase posterior do teste, o erro do tipo 1 é que requer atenção. A vantagem de usar técnicas automatizadas para determinar mutantes equivalentes é que as técnicas podem ser projetadas para encontrarem somente erros do tipo 1.

4.3.1 Descrição de Técnicas de Determinação de Mutantes Equivalentes

Reconhecer e eliminar mutantes equivalentes manualmente consome muito tempo e esforço, sendo essa tarefa um dos principais problemas para a aplicação

prática do teste de mutação. Nas próximas seções são descritos dois trabalhos cujo objetivo é determinar mutantes equivalentes automaticamente; para cada um dos trabalhos são apresentadas: as técnicas e ferramentas definidas.

4.3.1.1 Técnicas de Otimização de Compilador

Offut e Craft [OC94] apresentam seis técnicas de otimização de compilador para detectar mutantes equivalentes que foram definidas por Baldwin e Sayward [BS79]. A idéia principal utilizada nessa abordagem é que muitos mutantes equivalentes são, de algum modo, otimizações dos programas originais. Quando um mutante equivalente satisfaz uma regra de otimização do código, algoritmos podem detectar que de fato o mutante é equivalente. As seis técnicas são:

1. Determinação de código morto,
2. Propagação de constantes,
3. Propagação invariante,
4. Determinação de sub-expressão comum,
5. Determinação de loop invariante, e
6. Elevação e rebaixamento.

Offut e Craft construíram algoritmos para determinar classes de mutantes equivalentes baseados nas seis técnicas de otimização de compilador e na análise de fluxo de dados. A seguir são descritas as técnicas que foram adaptadas e aplicadas pelos autores.

- **Determinação de mutantes equivalentes através de código morto**

Nessa técnica são apresentados dois modos de matar código; o primeiro é através de comandos que nunca serão usados ou cuja execução é irrelevante. O código pode ser não alcançável ou *estático*, isso significa que não existe um caminho para o comando, ou *dinâmico*, onde não existe caminhos que possam ser executados. Através do grafo de fluxo de controle é possível verificar os nós não visitados e que representam código morto. Obviamente qualquer mutação

que altere um código morto não poderá afetar a saída do programa e é portanto equivalente.

O segundo modo de matar código é a definição morta, que é a definição de um dado que é redefinido antes de ser referenciado ou que nunca é referenciado. Qualquer mutação de um comando que tem uma definição morta será equivalente.

- ## Determinação de mutantes equivalentes usando propagação de constantes

A técnica denominada *propagação de constantes* envolve detectar definições cujos valores são constantes e que podem ser computados em tempo de compilação. A determinação de definições de constantes em um bloco é usada para determinar definições de constantes em outros blocos. Isso é realizado utilizando as informações derivadas da análise do fluxo de dados e de uma *tabela de constantes*, a qual tem uma entrada para cada definição de constante. Essa informação é usada para determinar mutantes equivalentes e ocorre quando um mutante não pode ser morto se uma variável tem o valor em sua entrada de definição de constantes na tabela. Depois desse procedimento terminar, a informação sobre cada definição armazenada nas constantes da tabela pode ser usada para determinar mutantes equivalentes. A atual implementação determina mutantes equivalentes dos tipos ABS, SVR, UOI, AOR, CSR, SCR, e ROR entre os 22 operadores existentes na ferramenta *Mothra*.

- ## Equivalência de mutantes usando propagação invariante

Uma *invariante* é uma relação entre duas variáveis ou entre uma variável e uma constante que devem ser verdadeiras entre dado ponto de um programa. Essas invariantes são separadas em duas categorias. O primeiro grupo de invariantes pertence às definições de variáveis contidas no programa e são armazenadas na *tabela de definições de invariantes*. O segundo tipo é um grupo mais geral que inclui uma invariante separada para cada comando do programa. Relacionamentos que são verdadeiros em um comando do programa são

armazenados na *tabela de comando invariantes* junto com o correspondente número do comando. Essa informação é usada para determinar mutantes equivalentes quando um mutante envolve uma variável que tem a invariante marcada na tabela de definições de invariantes. Por exemplo para matar um mutante de variável alterada, a nova variável deve ter um valor diferente da variável velha. Se a *tabela de decisão de definições invariantes* indicar que as duas variáveis são iguais, o mutante é equivalente.

- ## Determinação de mutantes equivalentes usando sub-expressões comuns

Otimização de compiladores reconhece sub-expressões comuns que freqüentemente surgem como variáveis temporárias durante o processo de compilação. Essa técnica é utilizada para reconhecer a equivalência entre duas variáveis.

O algoritmo construído para essa técnica estabelece duas tabelas, a primeira é uma *tabela de sub-expressão* onde as sub-expressões reconhecidas do código são mantidas. A segunda é a *tabela de definições de itens de dados*, que armazenam informações que referem-se a sub-expressões atribuídas para cada item de dado. Em outras palavras, a primeira tabela age como um tabela de variáveis temporárias e a segunda mantém um registro de quais variáveis temporárias são atribuídas para cada item de dado. Considere o comando de atribuição A = B + C – D, essas atribuições podem ser divididas nos seguintes comandos de atribuição usando as variáveis temporárias T1 e T2:

```
T1 = B + C
T2 = T1 - D
A  = T2
```

Nesse exemplo, a tabela de sub-expressão conterá entradas para T1 e T2. A tabela de definição de itens de dados deverá conter a informação de T2 sendo atribuída para o item de dado A.

Uma propriedade desse algoritmo garante que a tabela de sub-expressão para cada entrada é única. Em outras palavras, se uma sub-expressão é

encontrada no código que já existe na tabela, então a referência para a entrada existente é usada e a expressão redundante não é adicionada na tabela.

O objetivo da técnica de *determinação de sub-expressão comum* é determinar o relacionamento de igualdade entre as variáveis. Essa informação é então codificada para *invariantes* que são adicionadas para a *tabela de comandos invariantes*. Por esse motivo, essa técnica não determina mutante equivalente diretamente. Em vez disso, a informação obtida é usada em conjunto com os resultados da técnica de *propagação invariante* para determinar mutantes equivalentes.

- ## Determinação de mutantes equivalentes usando determinação de loop invariante

O operador de mutação *Do-loop* altera o limite de alcance dos *loops* através da alteração do rótulo do comando *Do*. Durante a otimização do código, o código que é invariante até o fim de um *loop* é freqüentemente movido para fora do *loop*, enquanto que na mutação o código pode ser movido para dentro ou para fora do *loop*. Se uma alteração de mutante altera o limite de um *loop* tal que um código invariante é movido para dentro ou fora do *loop*, então o mutante é equivalente. Essa técnica é usada para detectar se a mutação *DER (Do statement end replacement)*, que altera os limites de um *DO-loop*, é equivalente.

- ## Determinação de mutantes equivalentes usando elevação ou rebaixamento

Elevação ou rebaixamento visa a determinar se um código pode ser movido de uma situação onde era executado várias vezes para ser executado apenas uma vez, ou de situações onde aparece uma vez para uma situação onde aparece mais de uma vez. Por exemplo, se um comando será executado nas duas instruções de desvio de um comando *if-then-else*, e esse comando não depende de nenhum outro código do comando *if-then-else*, esse comando pode ser elevado e colocado antes do *if* ou rebaixado e colocado depois do *endif*.

4.3.1.2 Equalizador - Uma Ferramenta para Detecção de Mutantes Equivalentes

As seis técnicas mostradas nas seções anteriores, foram utilizadas na implementação da ferramenta de teste denominada *Equalizador*. Essa ferramenta é utilizada acoplada ao sistema de teste de mutação Mothra [Dem88, Cho89]. O Equalizador é implementado na linguagem de programação C e como a Mothra, trabalha com programas do Fortran-77. Através da ferramenta Mothra, programas em teste são analisados em uma linguagem intermediária chamada código intermediário Mothra (Mothra Intermediate Code - MIC) [KO91]. O *Equalizador* usa a tabela MIC para construir os grafos dos blocos básicos, encontrar todas as definições e encontrar os blocos básicos que cada definição alcança. Essa informação é passada separadamente para cada uma das quatro funções otimizadoras que criam tabelas indicando se o código morto é encontrado (Tabela de código morto), quais definições têm valores de constantes (Tabela de constantes), e quais comandos têm associações invariantes com elas (Tabelas de invariantes). As tabelas de invariantes tem informações sobre propagação de invariantes e determinação de subexpressões comuns.

Os autores utilizaram a ferramenta Equalizador em um experimento para determinar mutantes equivalentes em programas Fortran-77. Os programas inicialmente foram analisado manualmente para determinar os números corretos de mutantes equivalentes, e depois foi verificada a eficácia do *Equalizador* de acordo com o número de mutantes detectados. Em alguns casos, programas foram construídos para assegurar que o software trabalhava corretamente.

Offut e Craft observam que o potencial de determinação do *Equalizador* depende muito das características do programa que está sendo testado; por exemplo, para um dos programas do experimento foram determinados 49% dos mutantes equivalentes enquanto que para outro foi determinado apenas 1% dos mutantes equivalentes. A média obtida para os programas foi 10%.

4.3.1.3 Determinação Automática de Mutantes Equivalentes utilizando Restrições Matemáticas

O segundo trabalho descrito nesse capítulo foi definido por Offutt e Pan [OP97] e utiliza restrições matemáticas para matar mutantes equivalentes. Teste baseado em restrições (Constraint - Based Testing - CBT) utiliza restrições matemáticas para realizar teste [Off88]. Como restrições podem ser usadas para gerar casos de teste e satisfazer o teste de mutação [DMO91] os autores apresentam nesse trabalho o relacionamento existente entre essas técnicas.

Offutt e Pan [OP97] basearam-se no fato de que se um caso de teste mata um mutante, o sistema de restrição será satisfeito pelo caso de teste e que se o sistema de restrições não é verdadeiro, então não existe um caso de teste que poderá matar o mutante e consequentemente o mutante é equivalente. A abordagem geral utilizada segundo Offutt e Pan [OP97] é usar restrições para determinar mutantes equivalentes e verificar a não executabilidade em sistemas de restrições.

Os autores apresentam provas para três teoremas demonstrando como CBT pode ser utilizada para determinar mutantes equivalentes.

A estratégia geral para determinar mutantes equivalentes é encontrar contradições nos sistemas de restrições. Dado o fato que sistemas de restrições não executáveis é geralmente indecidível, o problema não pode ser resolvido completamente algoritmicamente. Entretanto, como mutantes equivalentes são determinados manualmente, soluções parciais são muito importantes.

A estratégia geral utilizada foi aplicar uma coleção de técnicas especiais de análise de casos e heurísticas para reconhecer sistemas de restrições não executáveis. Os casos observados foram os que ocorrem na maioria das mutações, baseando-se no fato que mutantes diferem de seus programas originais em pequenas e bem definidas alterações sintáticas.

A seguir são descritas e avaliadas três estratégias gerais que tentam reconhecer sistemas de restrições não executáveis, que são *negação, divisão de restrição e comparação de constantes.*

Negação

A estratégia de restrições é a técnica básica usada para reconhecer restrições não executáveis. Dada duas restrições, *negação* ou *negação parcial* é usada inicialmente para reescrever uma das duas restrições, então essas duas restrições são comparadas. Se elas são sintaticamente iguais, as restrições entram em conflito, e o sistema de restrições é não executável. Assim, um mutante com sistema de restrições não executável é equivalente. Exemplo, assuma duas restrições A e B, onde A é $(x+y) > z$ e B é $(x+y) \leq z$. A negação de A é $(x+y) \leq z$, denotada A'. Dado que A' e B são sintaticamente iguais, A e B entram em conflito.

Divisão de restrição

Divisão de restrição também é usada para reconhecer restrições não executáveis. Um caso que normalmente ocorre é uma restrição de necessidade como $(x+y) > 0$, junto com uma expressão de caminho tal como $(x < 0) \wedge (y < 0)$. A estratégia de negação não consegue reconhecer esse conflito e então a estratégia de divisão de restrição é usada. Dada duas restrições (ditas C e D), duas novas restrições (ditas A e B) são geradas tal que $C \Rightarrow A \vee B$. Então A e B são comparadas com D, e são realizadas provas mostrando que se A e B entram em conflito com D, então C entra em conflito com D.

Comparação de constantes

A terceira estratégia usa uma propriedade que é comum na criação de restrições para geração de casos de teste. A propriedade é completa, onde ambas as restrições têm o formato (V rop K), onde *V* é uma variável *rop* é um operador relacional, e *K* é uma constante. Adicionalmente, as variáveis em ambas as restrições devem ser as mesmas. Seja A a restrição $(X \text{ rop } K_1)$ e B a restrição $(X \text{ rop } K_2)$. Através da avaliação das duas constantes e operadores relacionais, nós podemos decidir se A entra em conflito com B. Essa estratégia é denominada comparação de constantes.

4.3.1.4 Equivalencer - Uma ferramenta para Detecção de Equivalência Baseada em Restrições

A ferramenta *Equivalencer* é integrada ao *Godzilla*, um gerador de dados de teste existente no conjunto de ferramentas de mutação Mothra [Dem88, Cho89]. Embora a técnica CBT seja independente da linguagem, Mothra trabalha com programas Fortran-77. *Equivalencer* foi implementada na linguagem de programação C.

Equivalencer utiliza-se de assertivas para ajudar a determinar mutantes equivalentes. Assertivas são restrições que um usuário insere em um programa em teste para restringir manualmente o domínio de entrada de algumas variáveis. Elas podem ser as pré-condições para o programa, predicados ou comandos específicos, ou predicados para uma entrada de função ou comandos específicos, ou ainda predicados para uma entrada de função ou programa.

Um estudo foi realizado utilizando *Equivalencer* para determinar mutantes equivalentes em 11 programas da linguagem FORTRAN-77. Cada programa foi analisado manualmente para determinar o verdadeiro número de mutantes equivalentes. A eficácia do *Equivalencer* tem sido comparada baseado na porcentagem dos mutantes equivalentes que foram determinados.

Embora *Equivalencer* tenha sido capaz de determinar em média aproximadamente a metade dos mutantes equivalentes, a porcentagem encontrada para cada programa variou consideravelmente; isso se deve a uma variedade de fatores, muitos deles limitações da própria ferramenta no qual *Equivalencer* está integrado.

Offutt e Pan [OP97] realizaram um estudo com 11 programas Fortran-77 e verificaram que a técnica de restrições matemáticas é aproximadamente cinco vezes mais rápida que a técnica de otimização de compiladores.

4.4 Considerações Finais

Nesse capítulo foram discutidos questões relacionadas a determinação de mutantes equivalentes, uma atividade que exige um grande esforço e a consequência é um custo alto para a aplicação do critério Análise de Mutantes.

Os principais resultados obtidos relacionados ao experimento de aplicação do critério Análise de Mutantes referentes aos mutantes equivalentes determinados foram apresentados. Os resultados observados contribuirão para um estudo futuro de automatização de determinação automática de mutantes equivalentes.

Alguns operadores que geraram o maior e o menor número de mutantes equivalentes e as porcentagens de mutantes equivalentes foram apresentadas. Sugere-se que a utilização ou não desses operadores seja avaliada pelo usuário que poderá estabelecer uma estratégia de aplicação para o critério Análise de Mutantes considerando os resultados aqui apresentados.

Também foi apresentado um resumo de técnicas que podem ser utilizadas para automatizar a determinação de mutantes equivalentes.

Capítulo 5

Conclusões e Trabalhos Futuros

O teste de software é uma das atividades mais onerosas no desenvolvimento de software e, para reduzir os custos associados ao teste, faz-se necessária a aplicação de técnicas e critérios que facilitem essa atividade e que mostrem indicações de como testar o software, e até quando testá-lo, fornecendo medidas que sejam confiáveis.

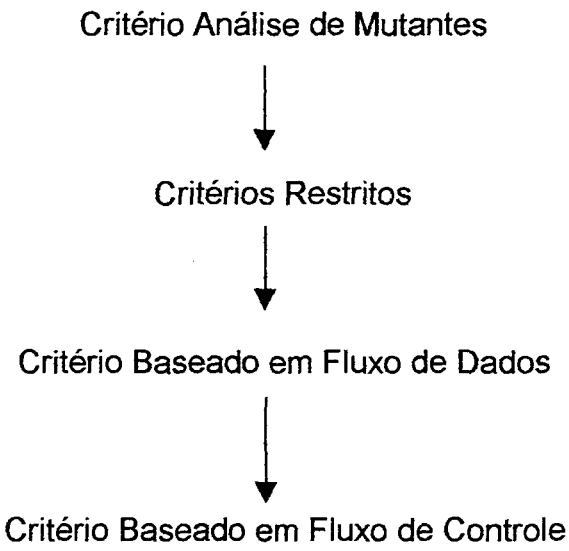
Para obter um teste de boa qualidade, as técnicas de teste devem ser aplicadas em conjunto; desse modo deve-se verificar qual estratégia de teste utilizar de modo a obter uma maior eficácia em revelar a presença de erros a um custo menor.

O critério Análise de Mutantes é um critério bastante atrativo para o teste de programas devido a sua habilidade na detecção de erros. Porém possui um alto custo computacional exigido na compilação e execução do grande número de mutantes gerados; outro problema é a determinação de mutantes equivalentes que necessita grande esforço e consome muito tempo.

Os Critérios Restritos, uma extensão para as diferentes famílias de critérios estruturais, permitem a utilização desses critérios juntamente com os princípios de técnicas de geração de dados de testes sensíveis a erros, aumentando a probabilidade dos dados gerados em revelar erros.

O trabalho nesta dissertação ressalta a relevância de estudos empíricos para estabelecer estratégias de teste que auxiliem a utilização das diversas técnicas e critérios de teste e apresenta resultados de um estudo empírico realizado com o objetivo principal de comparar o critério Todos-Potenciais-Usos Restritos com o critério Análise de Mutantes considerando os fatores: custo, em termos do número de casos de teste necessários, eficácia, em termos de número de erros revelados e *strength* (dificuldade de satisfação) de um critério dado que um outro também foi satisfeito.

Os estudos empíricos são importantes para estabelecer uma estratégia que auxilie a utilização das diversas técnicas e critérios de teste. Os resultados apresentados mostram que os Critérios Restritos constituem critérios intermediários entre os estruturais e o critério Análise de Mutantes em termos de custo, eficácia e *strength* (dificuldade de satisfação). Embora esses critérios sejam incomparáveis do ponto de vista teórico, os resultados apresentados nesse trabalho evidenciam a seguinte relação empírica:



Pode-se destacar como contribuições deste trabalho:

- Os resultados de avaliação de custo que mostram evidências que o critério Análise de Mutantes possui um maior custo de aplicação no que se refere ao número de casos de teste, exceto quando o programa em teste possui estruturas que podem aumentar o número de elementos requeridos para um critério estrutural: o programa *checkeq* exemplifica isso.
- Quanto à eficácia, os Critérios Restritos se mostraram mais eficazes que o correspondente critério estrutural porém menos eficazes que o critério Análise de Mutantes.
- A avaliação do *strength* (dificuldade de satisfação) do critério Análise de Mutantes em relação ao critério Todos-Potenciais-Usos Restritos mostrou que é mais fácil satisfazer o critério Análise de Mutantes tendo

sido satisfeito o critério Todos-Potenciais-Usos Restritos do que tendo sido somente satisfeito o seu correspondente critério estrutural Todos-Potenciais-Usos.

- Comparando-se os resultados obtidos, utilizando-se os conjuntos "ad-hoc" e aleatórios, têm-se que é importante satisfazer os critérios, isso implica numa maior cobertura para critérios que empiricamente incluem o critério sendo satisfeito, e pode implicar num maior número de erros revelados. Portanto, em casos onde a confiabilidade requerida é alta, é importante que o critério de teste seja satisfeito.

- A análise do strength (dificuldade de satisfação) do critério Análise de Mutantes considerando apenas o operador ORRN, com relação ao critério Todos-Potenciais-Usos Restritos foi realizada, dado o fato de que o operador ORRN assim como as restrições BOR/BRO utilizadas pelo critério Todos-Potenciais-Usos Restritos têm o objetivo de encontrar erros em predicados. Pôde ser verificado que as restrições BOR/BRO não incluem a mutação do operador relacional na presença de caminhos não executáveis.

- Uma possível estratégia de aplicação dos critérios foi apresentada e estabelece que os critérios sejam utilizados seguindo a relação empírica apontada pelos resultados obtidos e seguindo a confiabilidade requerida para o programa.

- Coleta de dados sobre os operadores da ferramenta Proteum que geraram o maior e o menor número de mutantes equivalentes no experimento com a qual foi verificado que se a estratégia incremental de aplicação dos critérios tal como descrita no Capítulo 3 for utilizada, os operadores que geraram o menor número de mutantes equivalentes, e que são incluídos na ferramenta Proteum para se garantir a inclusão do critério Todos-Ramos, podem ser desconsiderados, resultando na redução do numero de execuções do programa e conseqüentemente no custo do critério Análise de Mutantes.

- Uma revisão bibliográfica sobre os trabalhos que têm como objetivo determinação de mutantes equivalentes e que podem servir como base para automatizar essa atividade.

5.1. Trabalhos Futuros

Têm-se as seguintes perspectivas de continuação desse trabalho:

- Estudar a utilização de outros tipos de restrições para os Critérios Restritos visando a redução de custos, ou seja, verificar empiricamente quais tipos de restriçoes são mais adequadas para certos tipos de programas.
- Realizar comparações entre os critérios: Todos-Potenciais-Usos Restritos x Mutação Aleatória, e Todos-Potenciais-Usos Restritos x Mutação Restrita, para verificar o custo, eficácia e *strength*.
- Conduzir um experimento para validar a estratégia de aplicação dos critérios proposta no Capítulo 3.
- Com base nos resultados obtidos sobre mutantes equivalentes e também com base nas informações de outros estudos empíricos já realizados, estudar e implementar mecanismos para a determinação automática de mutantes equivalentes na ferramenta Proteum.

Os resultados a serem obtidos nesses trabalhos podem estabelecer formas alternativas, mais econômicas de testar o software, e mais eficazes se o tipo do programa em teste for considerado.

Referências Bibliográficas

[Acr80] A.T. Acree. *On Mutation.* PhD thesis. School of Information and Computer Science, Georgia Institute of Technology, August 1980.

[BA82] T.A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, Vol. 18(1):31-45, November 1982.

[BEK75] R.S. Boyer, B. Elspas, and N.L. Karl. Select: A formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, Vol. 10(6):234-245, June 1975.

[BP84] V.R. Basili and B.T. Pericone. Software error and complexity: An empirical investigation. *Communications of the ACM*, Vol. 27(1):42-52, January 1984.

[BS79] D. Baldwin and F. Sayward. *Heuristics for Determining Equivalence of Program Mutations.* CT, Res. Rep. 276, Department of Computer Science - Yale University, New Haven, 1979.

[Bud81] T.A. Budd - Mutation Analysis: *Ideas, Examples, Problems and Prospects, Computer Program Testing,* North-Holand Publishing Company, 1981.

[Cha91] M.L. Chaim. POKE-TOOL - *Uma ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados.* Dissertação de Mestrado, DCA/FEEC/Unicamp, Campinas - SP, Abril 1991.

[Cho89] B.J. Choi, et al., "The Mothra Tool Set", in Proceedings of the 22nd Hawaii International Conference on Systems and Software, Kona, Hawaii, January 1989.

[Cla76] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, Vol. SE-2(3):215-222, September 1976.

[Del93] Delamaro, M.E., "*Proteum - Um ambiente de teste baseado na Análise de Mutantes*", Dissertação de Mestrado, ICMSC/USP - São Carlos - SP, Outubro 1993.

[Dem78] R.A. De Millo. R.J. Lipton, F.G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer", IEEE Computer, April 1978.

[Dem87] R.A. De Millo. *Software Testing and Evaluation.* The Benjamin/Cummings Publishing Company, Inc, 1987.

[Dem88] R.A De Millo, et al. An Extended Overview of the Mothra Testing Environment, in Proc*, of the Second Workshop on Software Testing, Verification and Analysis,* Banff - Canada, 1988.

[Dem91] R.A De Millo and J. Offutt. Constrained-Based Automatic Test Data Generation, IEEE Transactions on Software Engineering, v. 17, n. 9, September 1991.

[DMM95] R.A De Millo and A.P. Mathur. A Grammar Based Fault Classification Scheme and its Application to the Classification of the Errors of text. Techical Report, CS/Purdue University, September 1995.

[Deu82] M.S. Deutsch. Software Verification and Validation, Englewwod Cliffs, Prentice-Hall, 1982.

[DN84] J.W. Duran and S.C. Ntafos. An evaluation of random testing. IEEE Transactions on Software Engineering, Vol. SE-10(4):438-444, July 1984.

[DMO91] R.A. De Millo and A.J. Offutt. *Constraint-based automatic test data generation. IEEE Transactions on Software Engineering,* Vol. SE-17(9):900-910, September 1991.

[Fra85] F.G. Frankl and E.J., Weyuker. "Data Flow Testing Tools", *in Proc. Softfair II*, San Francisco, CA, pp. 46-53, December 1985.

[Fra87] F.G. Frankl. *The use of Data Flow Information for the Selection and Evaluation of Software Test Data.* PhD Thesis, Department of Computer Science, New York University, New York, U.S.A., October 1987.

[Fri75] A.D. Friedman. Logical Design of Digital Systems. Computer Science Press, 1975.

[FW88] F.G. Frankl and E.J., Weyuker. An Applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, Vol. SE-14(10):1483-1498, October 1988.

[Fos80] K.A. Foster. "Error sensitive test cases analysis (ESTCA)," IEEE Transactions Software Engineering, Vol. SE-6, No. 3, 258-264. May 1980.

[GG75] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. IEEE Transactions on Software Engineering, Vol. SE-2(3):156-173, September 1975.

[HH85] D. Hedley and M.A. Hennell. The causes and effects of infeasible paths in computer programs. *In Proceeding of 8th. ICSE*, pages 259-266. UK, 1985.

[Hor92] J.R. Horgan, A.P Mathur. "Assessing Testing Tools in Research and Education", IEEE Software, vol. 9, Nº. , May 1992.

[How76] W. Howden. *Reliability of the path analysis testing strategies*. IEEE Transactions on Software Engineering, Vol. SE-2(3):208-215, September 1976.

[How77] W.E. Howden. Symbolic testing and dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, Vol. SE-3(4):266-278, July 1977.

[HT88] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *In Proc. of the Second Workshop on Software Testing, Verification and Analysis*, pages 206-215. Computer Science Press, Banff - Canada, July 1988.

[KO91] K.N. King and A. J. Offutt. A Fortran Language system for mutation-based software testing. Software-Practice and Experience, 21(7):685-718, July 1991.

[Kor90] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering,* Vol. SE-16(8):870-879, August 1990.

[Mal91] J.C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao*

Teste Estrutural de Software. Tese de Doutorado, DCA/FEEC/Unicamp, Campinas, SP, Julho 1991.

[Mat93] A.P. Mathur, W.E. Wong. "Evaluation of the Cost of Alternate Mutation Strategies", Anais do XII Simpósio Brasileiro de Engenharia de Software, pp.320-335, Rio de Janeiro, RJ, 1993.

[Mat94] A.P. Mathur, W.E. Wong. "An Empirical Comparison of Data Flow and Mutation-Based Test Adequacy Criteria". The Journal of Software Testing, Verification and Reliability, v. 4, n.1, pp. 9-31, 1994.

[McC76] T. McCabe. A software complexity measure. IEEE *Transactions on Software Engineering*, Vol. SE-2(4):308-320, December 1976.

[MCJ88] J.C. Maldonado, M.L. Chaim, and M. Jino. Seleção de casos de teste baseada nos critérios potenciais usos. *In II Simpósio Brasileiro de Engenharia de Software,* pages 24-35. Sociedade Brasileira de Computação - SBC, Canela - RS, Outubro 1988.

[Mye79] G.J. Myers. The Art of Software Testing, John Wiley & Sons, New York, 1979.

[MYV90] N. Malevris, D.F. Yates, and A. Veevers. Predictive metric for likely feasibility of program paths. *Information and Software Technology*, Vol. 32(2):115-118, March 1990.

[Nta84] S.C. Ntafos. "On Required Element Testing", *IEEE Transactions on Software Engineering*, SE-10(16), November 1984.

[Nta88] S.C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, SE-14(6):868-873, June 1988.

[OC94] A.J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *The Journal of Software Testing, Verification, and Reliability.* 4(3):131-154. September 1994.

[Off88] A.J. Offutt. Automatic Test Data Generation. PhD thesis, Georgia Institute of Technology, Technical report GIT-ICS 88/28. 1988.

[OJ97] A.J. Offutt and Jie Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths*. *The Journal of Software Testing, Verification,*

and Reliability. 7(3):165-192. September 1997.

[OW84] T.J. Ostrand and E.J. Weyuker, Collecting and categorizing software error data in an industrial environment. *The Journal of Systems and Software*, Vol. 4:289-300, 1984.

[Pre92] R.B. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, New-York, EUA, Third edition, 1992.

[Rap82] S. Rapps, E.J. Weyuker. *"Data Flow Analysis Techniques for Test Data Selection"*, In Proc. Int. Conf. Software Eng., Tokio, Japão, Set 1982.

[Rap85] S. Rapps, E.J. Weyuker. *"Selecting Software Test Data Using Data Flow Information"*, IEEE Trans. on Software Engineering, v.11 n.4, pp. 367-375, April, 1985.

[RSBC76] C.V. Ramamoorthy, F. H. Siu-Bun, and W.T. Chen. *On automated generation of program test data.* IEEE Transactions on Software Engineering, Vol. SE-2(4):293-300, December 1976.

[RW85] S. Rapps and E.J. Weyuker. *Selecting Software test using data flow information. IEEE Transactions on Software Engineering*, SE-11 (4):367-375, April 1985.

[Soa00a] I.W. Soares. *"Resultados de um Experimento de Avaliação dos Critérios de Teste: Todos-Potenciais-Usos Restritos e Análise de Mutantes"*, Relatório Técnico, UFPR, Curitiba - PR, Outubro 2000.

[Sou96] S.R.S. Souza. *Avaliação do Custo e Eficácia do Critério Análise de Mutantes na Atividade de Teste de Programas.* Dissertação de Mestrado, ICMSC/USP, São Carlos-SP, Junho 1996.

[Tai93] K.C. Tai. *Predicate-based teste generation for computer programs. In Proceedings of International Conference on Software Engineering*, pages 267-276. IEEE Press, May 1993.

[UY88] H. Ural and B. Yang. A Structural Test Selection Criterion. *Information Processing Letters*, 28(3): 157-163, July 1988.

[Ver92] S.R. Vergilio. *Caminhos não Executáveis: Caracterização, Previsão e*

Determinação para Suporte ao Teste de Programas. Dissertação de Mestrado, DCA/FEEC/Unicamp, Campinas – SP, Brasil, Janeiro 1992.

[Ver93] S.R. Vergilio, J.C. Maldonado, M. Jino. Uma Estratégia para Geração de Dados de Teste, *VII Simpósio Brasileiro de Engenharia de Software,* pp. 307-319, Rio de Janeiro, 1993.

[Ver97] S.R. Vergilio. *Critérios Restritos de Teste de Software: Uma Contribuição para Gerar Dados de Teste mais eficazes.* Tese de Doutorado, DCA/FEEC/Unicamp, Campinas, SP, Julho 1997.

[WC80] L.J. White and E.I. Cohen. *A domain strategy for computer program testing. IEEE Transactions on Software Engineering*, Vol. SE-6(3):247-257, May 1980.

[Wey84] E.J. Weyuker. The complexity of data flow criteria for test data selection. *Information Processing Letters*, 19(2):103-109, August 1984.

[Wey90] E.J. Weyuker. The cost of data flow testing: An empirical study. *IEEE Transactions on Software Engineering*, Vo. SE-16(2):121-128, February 1990.

[Wey93] E.J. Weyuker. More experience with data flow testing. *IEEE Transactions on Software Engineering*, Vol. SE-19(3):914-919, September 1993.

[WMM94] W.E. Wong, A.P. Mathur, and J.C. Maldonado. *Mutation versus all-uses: An Empirical evaluation of cost, strength and effectivenes.* In Software Quality and Productivity-Theory, Pratice, Education and Training. Hong Kong, December 1994.

[Won93] W.E. Wong. – *On Mutation and Data Flow.* PhD Thesis, Department of Computer Science, Purdue University, West Lafayette-IN, USA, December 1993.

[Won94a] W.E. Wong, et al. – Mutation versus All-uses: An Empirical Evaluation of Cost, Strength, and Effectiveness, *Software Quality and Productivity Theory, practice, education and training,* Hong Kong, December 1994.

Apêndice A

Operadores de Mutação da Ferramenta Proteum

Os operadores de mutação estão divididos em quatro grupos: **mutação de comandos, mutação de operadores, mutação de variáveis e mutação de constantes**. Ao todo são 71 operadores de mutação disponíveis na ferramenta Proteum. Na Tabela A.1 são apresentados os operadores e seus significados. Informações mais detalhadas sobre esses operadores podem ser encontradas em [Del93].

Tabela A.1 - Operadores de Mutação da Ferramenta Proteum

Operadores de Mutação de Comando	
Operador	**Descrição**
STRP	Força a execução de todos os comandos do programa
STRI	Força a execução de true e false em cada *if*
SSDL	Retira um comando de cada vez do programa
SRSR	Troca cada comando por todos os *returns* que existem na função em teste.
SGLR	Troca os comandos *goto* por todos os rótulos existentes na função.
SCRB	Troca do comando *continue* pelo comando *break*
SBRC	Troca o comando *break* pelo comando *continue* qdo é possível.
SBRn	Troca dos comandos *continue* ou *break* por uma função break_out_to_level_n(J) onde J varia de acordo com o número de laços aninhados. Essa função força a interrupção dos J laços externos.
SCRn	Troca dos comandos *continue* ou *break* por uma função break_out_to_level_n(J) onde J varia de acordo com o número de laços aninhados. Essa função força a transferência do programa para o final de J laços acima.
SWDD	Troca o comando *while* pelo comando *do-while*.
SDWD	Troca o comando *do-while* pelo comando *while*.

SMTT	Força a execução dos laços mais de uma vez.
SMTC	Interrompe a execução do laço após duas execuções.
SMVB	Move "}" para cima e para baixo, quando é possível.
SSWM	Força a execução de todos os cases do comando *switch*.
OAAA	Troca atribuição aritmética por outra atribuição aritmética.
OAAN	Troca operador aritmético por outro operador aritmético.
OABA	Troca atribuição aritmética por operador de atribuição bitwise.
OABN	Troca operador aritmético por outro operador bitwise.
OAEA	Troca atribuição aritmética por operador de atribuição plana.
OALN	Troca operador aritmético por operador lógico
OARN	Troca operador aritmético por operador relacional
OASA	Troca atrib. aritmética por operador atrib. de deslocamento
OASN	Troca oper. aritmético por oper. de deslocamento
OBAA	Troca atribuição *bitwise* por atribuição aritmética
OBAN	Troca operador *bitwise* por operador aritmético
OBBA	Troca atribuição *bitwise* por atribuição aritmética
OBBN	Troca operador *bitwise* por operador *bitwise*
OBEA	Troca atribuição *bitwise* por atribuição plana
OBLN	Troca operador *bitwise* por operador lógico
OBRN	Troca operador *bitwise* por operador relacional
OBSA	Troca atribuição *bitwise* por operador de atribuição de deslocamento
OBSN	Troca operador *bitwise* por operador de deslocamento
OEAA	Troca atribuição plana por atribuição aritmética
OEBA	Troca atribuição plana por atribuição *bitwise*
OESA	Troca atribuição plana por operador de atribuição de deslocamento.
OLAN	Troca operador lógico por operador aritmético
OLBN	Troca operador lógico por operador *bitwise*
OLLN	Troca operador lógico por outro operador lógico
OLRN	Troca operador lógico por operador relacional
OLSN	Troca operador lógico por operador de deslocamento
ORAN	Troca operador relacional por operador aritmético
ORBN	Troca operador relacional por operador *bitwise*
ORLN	Troca operador relacional por operador lógico
ORRN	Troca operador relacional por outro relacional
ORSN	Troca operador relacional por oper. de deslocamento

OSAA	Troca atrib. de deslocamento por atrib. aritmética
OSAN	Troca operador de deslocamento por oper. aritmético
OSBA	Troca atrib. de deslocamento por atrib. *bitwise*
OSBN	Troca operador de deslocamento por oper. *bitwise*
OSEA	Troca atrib. de deslocamento por atribuição plana
OSLN	Troca operador de deslocamento por oper. lógico
OSRN	Troca operador de deslocamento por oper. relacional
OSSA	Troca atrib. de deslocamento por outra de deslocam.
OSSN	Troca oper. de deslocamento por outro de deslocam.
Ouor	Troca operador de incremento/decremento.
OLNG	Insere negação lógica em condições compostas.
OCNG	Insere negação lógica.
OBNG	Insere negação no operador *bitwise*.
OIPM	Substitui os operadores de incremento/decremento que não possuem direção.
OCOR	Troca o tipo primitivo do operador *cast*.
Vsrr	Substitui as referências escalares por variáveis escalares, globais e locais do programa.
Varr	Substitui as referências a vetores por variáveis escalares, globais e locais do programa.
Vtrr	Substitui as referências a estruturas e uniões por variáveis escalares, globais e locais do programa.
Vprr	Substitui as referências a apontadores por variáveis escalares, globais e locais do programa.
VSCR	Troca as referências a componentes de uma estrutura por demais componentes da mesma estrutura.
VDTR	Força cada referência escalar possuir cada um dos valores: negativo, positivo e zero.
VTWD	Substitui referência escalar pelo seu valor sucessor e antecessor.
CRCR	Troca constantes por: 0,1,-1, dependendo do tipo de referência.
Cccr	Troca constantes por todas as constantes do programa .
Ccsr	Troca referências escalares por constantes.

Apêndice B

Dados Coletados - Programa Cal

Este apêndice mostra um exemplo dos dados coletados durante a condução do experimento apresentado no Capítulo 4 para o programa *Cal*. Os dados coletados para os outros programas do experimento podem ser encontrados em Soares [Soa00a].

A Tabela B.1 - Apresenta as informações obtidas através da aplicação dos conjuntos de casos de teste "ad-hoc" e aleatórios eficazes para os critérios Todos-Potenciais-Usos (PU), Todos-Potenciais-Usos Restritos (PU-R) e submetidos ao critério Análise de Mutantes (AM). As informações referem-se a:

- Números de casos de teste dos conjuntos "ad-hoc" e aleatórios;
- Números de casos de teste eficazes para o critério AM;
- Números de mutantes gerados, mortos e equivalentes;
- Cobertura Real (excluindo os mutantes equivalentes) e Cobertura Geral (incluindo os mutantes equivalentes).

A Tabela B.2 - Apresenta as informações obtidas através da aplicação dos conjuntos de casos de teste "ad-hoc" e aleatórios, e os casos adicionais necessários para satisfazer o critério Análise de Mutantes (AM).

Tabela B.1 - Escore de Mutação obtido para o critério AM através dos conjuntos de casos de teste "ad-hoc" e aleatórios

Conjuntos de casos de teste	**Critério**	**N.° de casos de teste**	**N.° de casos de teste eficazes**	**Número de Mutantes**			**Cobertura**	
				Gerados	**Mortos**	**Equivalentes**	**Real**	**Geral**
$Ct_{ad\text{-}hoc}$	PU	12	12	4324	3871	309	0.90	0.97
$Ct_{ad\text{-}hoc}$	PU-R	17	17		3966		0.92	0.99
Ct_{aleat}	PU	7	7		3578		0.83	0.90
Ct_{aleat}	PU-R	17	17		3942		0.91	0.98

Tabela B.2 - Escore de Mutação final obtido para o critério AM

Conjuntos de casos de teste						Número de Mutantes			Cobertura	
$Ct_{ad\text{-}hoc}$	Ct_{aleat}	$Ct_{ad\text{-}hoc+aleat}$	Eficazes	Adic.	Tot.	Gerados	Mortos	Equivalentes	Real	Geral
37	162	199	27		27	4324	3977	309	0.92	0.99
			27	5	32		4015		0.93	1.00

A seguir são apresentadas as tabelas que apresentam os resultados da análise da eficácia dos dados de teste que contribuíram para a cobertura do critério Todos-Potenciais-Usos, Todos-Potenciais-Usos Restritos e Análise de Mutantes para o primeiro conjunto de programas incorretos gerados por Wong [WMM94] para o programa *Cal*.

As Tabelas B.3, B.4, B.5, B.6 e B.7 apresentam os resultados para o programa *cal*, o conjunto de versões incorretas é composto por 20 (vinte) programas.

Tabela B.3 - Eficácia Conjunto 1 - PU - Geração "ad-hoc" ($Ct_{ad\text{-}hoc\ -\ PU}$)

Test Case 1	1
Test Case 2	1 3 4 7 8 9 10 11 12 13 14 15 16 17 18
Test Case 3	1 3 4 7 8 13 14 16 18 19
Test Case 4	6 7 8 9 10 11 12 13 14 15 16 17 20
Test Case 5	6 7 8 9 10 11 13 14 15 16 17 18 19 20
Test Case 6	1 2 4 7 8 12 13 14 15 16 17 18 19
Test Case 14	1
Test Case 18	
Test Case 21	1 3 4 7 8 10 12 13 14 15 16 17 19
Test Case 25	1 3 4 7 8 13 14 16 17
Test Case 26	1 3 4 7 8 10 13 14 15 16 17
Test Case 27	1 3 4 7 8 10 13 14 15 16 17

Tabela B.4 - Eficácia Conjunto 1 - PU-R - Geração "ad-hoc" ($Ct_{ad\text{-}hoc\ -\ PU\text{-}R}$)

Test Case 1	1
Test Case 2	1 3 4 7 8 9 10 11 12 13 14 15 16 17 18
Test Case 3	1 3 4 7 8 13 14 16 18 19
Test Case 4	6 7 8 9 10 11 12 13 14 15 16 17 20
Test Case 5	6 7 8 9 10 11 13 14 15 16 17 18 19 20
Test Case 6	1 2 4 7 8 12 13 14 15 16 17 18 19
Test Case 7	1 3 4 7 8 10 11 12 13 14 15 16 18 19
Test Case 13	1
Test Case 14	1
Test Case 18	
Test Case 21	1 3 4 7 8 10 12 13 14 15 16 17 19
Test Case 24	1 3 4 7 8 9 10 11 12 13 14 15 16 17 18 19
Test Case 25	1 3 4 7 8 13 14 16 17
Test Case 26	1 3 4 7 8 10 13 14 15 16 17
Test Case 27	1 3 4 7 8 10 13 14 15 16 17
Test Case 35	5
Test Case 37	1 3

Tabela B.5 - Eficácia Conjunto 1- PU - Geração Aleatória ($Ct_{aleat - PU}$)

Test Case 1	1 2 4 7 8 13 14 16 17 18
Test Case 2	5
Test Case 4	1
Test Case 19	6 7 8 9 10 11 12 13 14 15 16 17
Test Case 20	1
Test Case 21	1 3 4 7 8 13 14 16 17
Test Case 23	1 3 4 7 8 13 14 16 17

Tabela B.6 - Eficácia Conjunto 1 - PU-R - Geração Aleatória ($Ct_{aleat - PU-R}$)

Test Case 1	1 2 4 7 8 13 14 16 17 18
Test Case 2	5
Test Case 4	1
Test Case 19	6 7 8 9 10 11 12 13 14 15 16 17
Test Case 20	1
Test Case 21	1 3 4 7 8 13 14 16 17
Test Case 23	1 3 4 7 8 13 14 16 17
Test Case 27	1 3 4 7 8 13 14 16 17
Test Case 30	6 7 8 9 10 11 12 13 14 15 16 17 20
Test Case 31	1
Test Case 39	1 3
Test Case 40	1 3 4 7 8 10 11 13 14 15 16 17
Test Case 46	1 3 4 7 8 10 13 14 15 16 17
Test Case 52	1
Test Case 61	6 7 8 10 13 14 15 16 17 20
Test Case 69	1
Test Case 95	

Tabela B.7 - Eficácia Conjunto 1 - AM - Geração "ad-hoc", aleatória e manual (Ct_{AM})

Test Case 1 ($Ct_{ad\text{-}hoc\ PU\text{-}1}$)	1
Test Case 2 ($Ct_{ad\text{-}hoc\ PU\text{-}2}$)	1 3 4 7 8 9 10 11 12 13 14 15 16 17 18
Test Case 3 ($Ct_{ad\text{-}hoc\ PU\text{-}3}$)	1 3 4 7 8 13 14 16 18 19
Test Case 4 ($Ct_{ad\text{-}hoc\ PU\text{-}4}$)	6 7 8 9 10 11 12 13 14 15 16 17 20
Test Case 5 ($Ct_{ad\text{-}hoc\ PU\text{-}5}$)	6 7 8 9 10 11 13 14 15 16 17 18 19 20
Test Case 6 ($Ct_{ad\text{-}hoc\ PU\text{-}6}$)	1 2 4 7 8 12 13 14 15 16 17 18 19
Test Case 7 ($Ct_{ad\text{-}hoc\ PU\text{-}7}$)	1 3 4 7 8 10 11 12 13 14 15 16 18 19
Test Case 8 ($Ct_{ad\text{-}hoc\text{-}8}$)	1 3 7 10 13 14 15 16 19
Test Case 9 ($Ct_{ad\text{-}hoc\text{-}9}$)	1 3 7 8 13 14
Test Case 10 ($Ct_{ad\text{-}hoc\text{-}12}$)	1 3 4 7 12 13 14 15 16 17 18 19
Test Case 11 ($Ct_{ad\text{-}hoc\ PU\text{-}R\text{-}13}$)	1
Test Case 12 ($Ct_{ad\text{-}hoc\ PU\text{-}18}$)	
Test Case 13 ($Ct_{ad\text{-}hoc\text{-}19}$)	1
Test Case 14 ($Ct_{ad\text{-}hoc\ PU\text{-}21}$)	1 3 4 7 8 10 12 13 14 15 16 17 19
Test Case 15 ($Ct_{ad\text{-}hoc\ PU\text{-}R\text{-}24}$)	1 3 4 7 8 9 10 11 12 13 14 15 16 17 18 19
Test Case 16 ($Ct_{ad\text{-}hoc\ PU\text{-}25}$)	1 3 4 7 8 13 14 16 17
Test Case 17 ($Ct_{ad\text{-}hoc\ PU\text{-}26}$)	1 3 4 7 8 10 13 14 15 16 17
Test Case 18 ($Ct_{ad\text{-}hoc\ PU\text{-}27}$)	1 3 4 7 8 10 13 14 15 16 17
Test Case 19 ($Ct_{ad\text{-}hoc\text{-}28}$)	1 3 4 7 8 9 10 11 13 14 15 16 17 18
Test Case 20 ($Ct_{ad\text{-}hoc\text{-}30}$)	1 3 4 7 8 10 11 12 13 14 15 16 17
Test Case 21 ($Ct_{ad\text{-}hoc\ PU\text{-}R\text{-}35}$)	5
Test Case 22 ($Ct_{aleat\ PU\text{-}4}$)	4
Test Case 23 ($Ct_{aleat\ -12}$)	

Test Case 24 ($Ct_{aleat\ PU-31}$)	1
Test Case 25 ($Ct_{aleat\ PU-R-40}$)	1 3 4 7 8 10 11 13 14 15 16 17
Test Case 26 ($Ct_{aleat\ -58}$)	6 7 8 9 10 11 12 13 14 15 16 17 20
Test Case 27 ($Ct_{aleat\ PU-R-69}$)	1
Test Case 28 ($Ct_{atual\ -1}$)	6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Test Case 29 ($Ct_{atual\ -2}$)	
Test Case 30 ($Ct_{atual\ -3}$)	6 7 8 9 10 11 13 14 15 16 17 20
Test Case 31 ($Ct_{atual\ -4}$)	1
Test Case 32 ($Ct_{atual\ -5}$)	1 3 4 7 8 13 14 16 17

Nas tabelas seguintes serão apresentados os dados obtidos do segundo conjunto de programas incorretos para verificar a eficácia dos critérios, esse conjunto foi gerado no experimento atual, para cada programa em teste foram geradas dez versões incorretas.

As Tabelas B.8, B.9, B.10, B.11 e B.12 apresentam os resultados para o programa *cal.*

Tabela B.8 - Eficácia Conjunto 2- *cal* - PU - Geração "ad-hoc" ($Ct_{ad\text{-}hoc\ -\ PU}$)

Test Case 1	
Test Case 2	1 4 5 6 7 8 9
Test Case 3	1 4 5 6 7 8 9
Test Case 4	1 4 5 6 7
Test Case 5	1 4 5 6 7 8 9
Test Case 6	1 2 4 5 6 7 8 9
Test Case 14	3
Test Case 18	
Test Case 21	1 4 5 6 7
Test Case 25	1 4 5 6 7
Test Case 26	1 4 5 6 7
Test Case 27	1 4 5 6 7

Tabela B.9 - Eficácia Conjunto 2- *cal* - PU-R - Geração "ad-hoc" ($Ct_{ad\text{-}hoc\ -\ PU\text{-}R}$)

Test Case 1	
Test Case 2	1 4 5 6 7 8 9
Test Case 3	1 4 5 6 7 8 9
Test Case 4	1 4 5 6 7
Test Case 5	1 4 5 6 7 8 9
Test Case 6	1 2 4 5 6 7 8 9
Test Case 7	1 4 5 6 7 8 9
Test Case 13	
Test Case 14	3
Test Case 18	
Test Case 21	1 4 5 6 7
Test Case 24	1 4 5 6 7
Test Case 25	1 4 5 6 7
Test Case 26	1 4 5 6 7
Test Case 27	1 4 5 6 7
Test Case 35	10
Test Case 37	3

Tabela B.10 - Eficácia Conjunto 2- *cal* - PU- Geração Aleatória ($Ct_{Aleat - PU}$)

Test Case 1	1 2 4 5 6 7 8 9
Test Case 2	10
Test Case 4	
Test Case 19	1 4 5 6 7
Test Case 20	3
Test Case 21	1 4 5 6 7
Test Case 23	1 4 5 6 7

Tabela B.11 - Eficácia Conjunto 2- *cal* - PU-R Geração Aleatória ($Ct_{Aleat - PU-R}$)

Test Case 1	1 2 4 5 6 7 8 9
Test Case 2	10
Test Case 4	
Test Case 19	1 4 5 6 7
Test Case 20	3
Test Case 21	1 4 5 6 7
Test Case 23	1 4 5 6 7
Test Case 27	1 4 5 6 7
Test Case 30	1 4 5 6 7
Test Case 31	3
Test Case 39	3
Test Case 40	1 5 6 7
Test Case 46	1 5 6 7
Test Case 52	3
Test Case 61	1 4 6 7
Test Case 69	
Test Case 95	

Tabela B.12 - Eficácia Conjunto 2- *cal* - AM - Geração "ad-hoc", aleatória e manual (Ct_{AM})

Test Case 1	($Ct_{ad\text{-}hoc\ PU\text{-}1}$)	
Test Case 2	($Ct_{ad\text{-}hoc\ PU\text{-}2}$)	1 4 5 6 7 8 9
Test Case 3	($Ct_{ad\text{-}hoc\ PU\text{-}3}$)	1 4 5 6 7 8 9
Test Case 4	($Ct_{ad\text{-}hoc\ PU\text{-}4}$)	1 4 5 6 7
Test Case 5	($Ct_{ad\text{-}hoc\ PU\text{-}5}$)	1 4 5 6 7 8 9
Test Case 6	($Ct_{ad\text{-}hoc\ PU\text{-}6}$)	1 2 4 5 6 7 8 9
Test Case 7	($Ct_{ad\text{-}hoc\ PU\text{-}7}$)	1 4 5 6 7 8 9
Test Case 8	($Ct_{ad\text{-}hoc\text{-}8}$)	1 4 5 6 7 8 9
Test Case 9	($Ct_{ad\text{-}hoc\text{-}9}$)	1 4 5 6 7
Test Case 10	($Ct_{aleat\text{-}12}$)	1 4 5 6 7 8 9
Test Case 11	($Ct_{ad\text{-}hoc\ PU\text{-}R\text{-}13}$)	
Test Case 12	($Ct_{ad\text{-}hoc\ PU\text{-}18}$)	
Test Case 13	($Ct_{ad\text{-}hoc\text{-}19}$)	3
Test Case 14	($Ct_{ad\text{-}hoc\ PU\text{-}21}$)	1 4 5 6 7
Test Case 15	($Ct_{ad\text{-}hoc\ PU\text{-}R\text{-}24}$)	1 4 5 6 7
Test Case 16	($Ct_{ad\text{-}hoc\ PU\text{-}25}$)	1 4 5 6 7
Test Case 17	($Ct_{ad\text{-}hoc\ PU\text{-}26}$)	1 4 5 6 7
Test Case 18	($Ct_{ad\text{-}hoc\ PU\text{-}27}$)	1 4 5 6 7
Test Case 19	($Ct_{ad\text{-}hoc\text{-}28}$)	1 4 5 6 7 8 9
Test Case 20	($Ct_{ad\text{-}hoc\text{-}30}$)	1 4 5 6 7
Test Case 21	($Ct_{ad\text{-}hoc\ PU\text{-}R\text{-}35}$)	10
Test Case 22	($Ct_{aleat\ PU\text{-}4}$)	1
Test Case 23	($Ct_{ad\text{-}hoc\ -12}$)	
Test Case 24	($Ct_{aleat\ PU\text{-}31}$)	3
Test Case 25	($Ct_{aleat\ PU\text{-}R\text{-}40}$)	1 5 6 7
Test Case 26	($Ct_{aleat\ -58}$)	1 4 5 6 7

Test Case 27 ($Ct_{aleat\ PU\text{-}R\text{-}69}$)	
Test Case 28 ($Ct_{atual\ -1}$)	1 4 5 6 7 8 9
Test Case 29 ($Ct_{atual\ -2}$)	
Test Case 30 ($Ct_{atual\ -3}$)	1 4 5 6 7
Test Case 31 ($Ct_{atual\ -4}$)	
Test Case 32 ($Ct_{atual\ -5}$)	1 4 5 6 7

A seguir é apresentado o relatório gerado pela Proteum onde está inserido o conjunto AM-adequado para o programa Cal. O relatório possui as seguintes informações: nome da sessão de teste, total de mutantes, número de mutantes executados, anômalos, ativos, vivos e equivalentes, escore de mutação, operadores de mutação com suas respectivas porcentagens, e os casos de teste eficientes com o número de mutantes mortos por cada um.

Arquivo Cal-9.lst

```
PROGRAM TESTE: cal-9
SOURCE FILE: cal
TOTAL MUTANTS: 4324
ANOMALOUS MUTANTS: 0
ACTIVE MUTANTS: 4324
ALIVE MUTANTS: 0
EQUIVALENT MUTANTS: 309
MUTATION SCORE:  1.00
OPERATORS:
Cccr 100%   Ccsr 100%   CRCR 100%   OAAA 100%
OAAN 100%   OABA 100%   OABN 100%   OAEA 100%
OALN 100%   OARN 100%   OASA 100%   OASN 100%
OBAA 100%   OBAN 100%   OBBA 100%   OBBN 100%
OBEA 100%   OBLN 100%   OBNG 100%   OBRN 100%
OBSA 100%   OBSN 100%   OCNG 100%   OCOR 100%
OEAA 100%   OEBA 100%   OESA 100%   Oido 100%
OIPM 100%   OLAN 100%   OLBN 100%   OLLN 100%
OLNG 100%   OLRN 100%   OLSN 100%   ORAN 100%
ORBN 100%   ORLN 100%   ORRN 100%   ORSN 100%
OSAA 100%   OSAN 100%   OSBA 100%   OSBN 100%
OSEA 100%   OSLN 100%   OSRN 100%   OSSA 100%
OSSN 100%   SBRC 100%   SBRn 100%   SCRB 100%
SCRn 100%   SDWD 100%   SGLR 100%   SMTC 100%
SMTT 100%   SMVB 100%   SRSR 100%   SSDL 100%
SSWM 100%   STRI 100%   STRP 100%   SWDD 100%
```

▯ Varr 100% VDTR 100% Vprr 100% VSCR 100%
▯ Vsrr 100% Vtrr 100% VTWD 100%
▯
▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯

Teste Case # 1
Number of Dead Mutants: 146

Teste Case # 2
Number of Dead Mutants: 1908

Teste Case # 3
Number of Dead Mutants: 67

Teste Case # 4
Number of Dead Mutants: 1141

Teste Case # 5
Number of Dead Mutants: 43

Teste Case # 6
Number of Dead Mutants: 213

Teste Case # 7
Number of Dead Mutants: 31

Teste Case # 8
Number of Dead Mutants: 2

Teste Case # 9
Number of Dead Mutants: 8

Teste Case # 12
Number of Dead Mutants: 9

Teste Case # 13
Number of Dead Mutants: 25

Teste Case # 15
Number of Dead Mutants: 61

Teste Case # 16
Number of Dead Mutants: 56

Teste Case # 17
Number of Dead Mutants: 4

Teste Case # 18
Number of Dead Mutants: 1

Teste Case # 19
Number of Dead Mutants: 22

Teste Case # 20
Number of Dead Mutants: 2

Teste Case # 21
Number of Dead Mutants: 202

Teste Case # 22
Number of Dead Mutants: 5

Teste Case # 24
Number of Dead Mutants: 1

Teste Case # 25
Number of Dead Mutants:

Teste Case # 28
Number of Dead Mutants: 1

Teste Case # 33
Number of Dead Mutants: 3

Teste Case # 40
Number of Dead Mutants: 1

Teste Case # 48
Number of Dead Mutants: 3

Teste Case # 64
Number of Dead Mutants: 2

Teste Case # 75
Number of Dead Mutants: 2

Teste Case # 76
Number of Dead Mutants: 6

Teste Case # 77
Number of Dead Mutants: 24

Teste Case # 78
Number of Dead Mutants: 1

Teste Case # 79
Number of Dead Mutants: 1

Teste Case # 80
Number of Dead Mutants: 6